Table of Contents

Table of Contents

Table of Contents

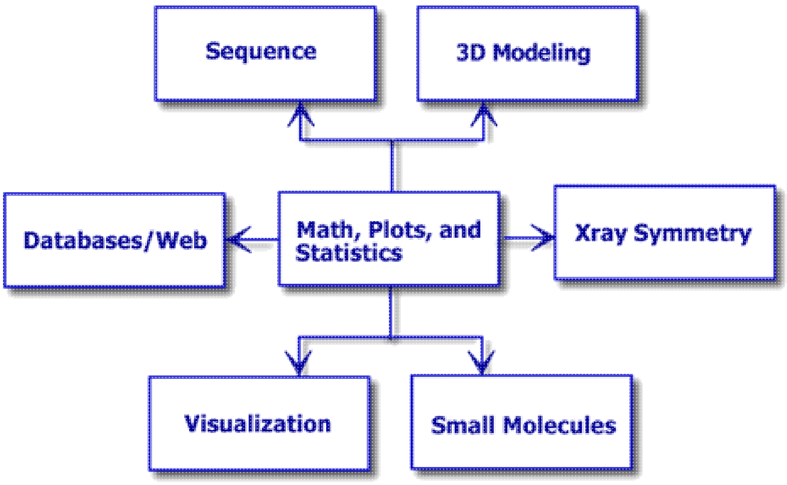
Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction



ICM stands for Internal Coordinate Mechanics and was first designed and built to predict low energy conformations of biomolecules. ICM also is a programming environment for various tasks in computational structural biology, sequence analysis and rational drug design. The original goal was to develop algorithms for energy optimization of several biopolymers with respect to an arbitrary subset of *internal coordinates* such as bond lengths, bond angles torsion angles and phase angles. The efficient and general global optimization method which evolved from the original ICM method is still the central piece of the program. It is this basic algorithm which is used for peptide prediction, homology modeling and loop simulations, flexible macromolecular docking and energy refinement. However the complexity of problems related to structure prediction and analysis, as well as the desire for perfection, compactness and consistency, led to the program's expansion into neighboring areas such as graphics, chemistry, sequence analysis and database searches, mathematics, statistics and plotting.
The original meaning became too narrow, but the name was kept. The current integrated ICM shell contains hundreds of variables, functions, commands, database and web tools, novel algorithms for structure prediction and analysis into a powerful, yet compact program which is still called ICM. The seven principal areas are centered around a general core of shell-language and data analysis and visualization.

Release notes

In this section we keep track of all the latest changes in different modules of ICM.

Version 3.8-1 Mar 13 2014

- Added search in PubChem from the general search toolbar
- `sort table column` is added
- fixed repaint issue of the frozen column in the table.
- _molScreen script and corresponding KNIME node is added. (interface to 2D fingerprint and 3D pharmacophore models)
- `SAR analysis` command is added.
- `R-group decomposition` supports R-groups with more than one heavy neighbor. (e.g: R1 in a ring)
- Peptide builder from extended list of amino acids
- Fixed crash on Linux when locale set to some non-English (e.g: de_LU.utf8)

Version 3.8-0

- Export of chemical structures, alignments and plots to Windows Meta-file (WMF) now has anti-aliased fonts and smooth line connections.
- Fixed crash on `compress grob` command
- Anaglyph stereo mode is added.
- added interface to BLAST search at NCBI (main Search bar in GUI)
- added Extended Connectivity Fingerprints (ECFP)
- Fixed bug with table selection in macros when table with the same name exists globally.
- xstick transparency mask, atom and residue font size and family now are stored in slides.
- Ctrl +/- in the table view resize the cell size (in addition to font size)
- Chemical Search dialog allows to select an existing chemical table for Molcart or local search.
- base html path is set correctly for commands like: read html "http://google.com"
- Chemistry/Full Model Panel is added

- Added macro and GUI to export object's stack as a movie. (Right click on the object: Tools/Export Stack")
- Covalent docking in "batch" mode in the ligand editor.
- "Find in alignment" supports wildcard patterns: N?[TS]
- Added confidence interface visualisation in table plots. (see `make plot` command and GUI dialog)
- InChI support. `InChi`(X_chemical [key]), Chemical(S_inchi)
- `Name`(user) returns OS user name.
- "copy off" option for `set field` command to prevent field from being copied with `copy` object command.
- Added macro and GUI to export slides as a movie
- added ToxScore function into Insert Column Dialog
- fixed several issues with sugar molecules representation in the workspace
- recursive SMARTS of more than one level are supported.
- row heights are preserved in ICB file
- added GUI interface to add image column and modify its contents
- Fixed crash when deleting 2D label which was used in slides.
- `set type mmff` on the converted object does not change aromatic bond type to pure aromatic
- added GUI.font
- added support for recursive SMARTS in `R-group decomposition` (e.g: [R1;$(CCN)])
- `Distance chemical` supports `nProc`
- Fixed a crash in 2D label edit dialog (image list)
- Image column is exported to Excel.
- Added the ability to add custom functions into 'Insert Column' dialog. (see `gui programming` for details)
- Dynamic grouping by column
- _chemSuper preserves original columns
- Combined "Pretty" view from docking hit list and Ligand Editor.
- `set site alignment` and `delete site alignment` are added
- added simple=S_cols option for `read table mol` to prevent type guessing (keep column as `sarray`)
- fixed some issues with `HTTP.proxy`
- `write image alignment` and `set property alignment` commands
- added support for `STL` mesh format (popular for 3D printing). Example: write grob g_mesh "mesh.stl"
- PubChem CID lookup function and Insert column of pubchem CIDs into a chemical table

Version 3.7-3b

- added search in POCKETOME option to the main search bar
- added clusterTableApf macro
- Wire color is not saved in ICB file. FIXED
- Wire thickness is not saved in ICB file. FIXED
- Fixed inability to drag distance labels after changing font.
- added R/S labels display in the 3D ligand editor
- added optional normal vector argument to `Grob`("ELLIPSOID"|"TORUS"|"CYLINDER" ...)
- plot labels now support '\n'
- fixed bug in title label for histogram plot with multiple series.
- icm.gui 'REQUIRED' checks for empty selection
- KNIME nodes v1.01 (Ligand selection in MolDockPrep module, new options in the MolDock module)
- Added Kernel Regression model. (`learn{ lean type="nn" })
- added support for MOL V3000 extension format.
- optimized core replacement search speed
- added `blob` support in the shell
- UTF8 encoding support in the terminal
- Column format dialog works with column selection.
- OpenMP support `nProc` variable to toggle internal parallelization of ICM commands.
- Fixed bug in `read grob` from URL
- Fixed bug in write matrix into ICB file (if rows were reordered)
- `Collection( table )` and `Table( collection )` were added
- Added the ability to use ICM built-in functions in the *function* section of the `add column function` command
- makeIndexPdb macro also builds PDB ligand table which can be search by substructure from the PDB search bar (just paste smiles or use molecular editor)
- Added COLOR preferences (accessible through GUI Preferences dialog)
- Detachable tabs with tables, alignments and other objects. (Drag the tab outside and it will become a separate window).

- Index(alig as_)
- alig[I_index] non-continuous sub-alignment
- added support for optional ph4 features and constraints
- added 'onSelectionChange' object named field to assign actions for selection changes in 3D
- added ability to link mesh to the object for simultaneous rotation, superimpose and display.
- added scanFilterHitlist macro to quickly filter large docking hit lists for top scored hits. (Integrated into main scanMakeHitList macro and corresponding GUI dialog)
- optimized `read object` for big files.
- added $ support in index expression. (refer to the last element of the array)
- Fixed bug in `Date` to `String` conversion. (week of the day and related fields are OK now)
- Added conditional expression support a?b:c
- Added 'delete query' option to `modify chemical` command to clear SMARTS search attributes.
- logical expression shortcuts: if (Nof(A)>=5 & A[5] == 1) ...
- added 'term' option in macro declaration: saves and restores energy terms after return from the macro
- balloon tooltip for cluster nodes (Right Click/Change Record Labels to specify list of columns in the dialog)
- better docking convergence in the Ligand Editor
- LIGAND.displayDockingMoves is set to 'no' by default
- new Area/Volume distribution for `icmPocketFinder` plot
- comments for stack conformations. (`set comment store conf` Name `conf`)
- Initial stack of ligand positions before docking in the ligand editor
- 'MolArea' model is added to calculate accessible surface are for 2D chemical. Predict(Chemical("CCCCO") "MolArea")
- File.ICM Session.New/Clone added.
- JavaScript errors are printed to the terminal
- added `Image( X_Chemical )`
- covalent ligand support in the ligand editor.
- added triple-quote string constants (arbitrary length and content text can be easily used as string constants)
- `make tree object` fills "ORDER" atom field which corresponds to the original atom order. (can be used to assign various properties)
- added `Chemical`(R_react) and Chemical(X_chem, reaction) to convert reaction to chemical and back
- `write image chemical` supports HTML5 canvas export and does not require X11 connection on Linux anymore (can be used in various batch scripts, CGIs, etc.)
- added `Info model` function to return prediction model properties.
- added `Split`([i_minSize] [i_maxSize] bond) to split molecule into a various fragments
- speed improvements in `enumerate tautomer` command
- Added `String`(macro all) to return macro source and all dependent macros.
- Added `String`(X_chem html) `String`(w_img html) to export chemicals and images into in-line html representation
- HTML5 canvas support for 2D drawing. (chemicals,plots,alignments,etc. ...)
- `read json` now accepts entries with multiple roots. (array on the top level)
- XML parser treat comments and CDATA correctly.
- added `Select by alignment` function
- fixed bug in select-alignment
- allow to use ICM user-defined functions in `add column function`
- _dockScan script is KNIME compatible. Added _dockProjPrep script to prepare docking project in the KNIME work-flow.
- new sugar view in the workspace.
- `assign residue` command
- Easy selection finder in the alignment ('Highlight Selection' checkbox in the tools panel)
- Balloon popup over plot elements. ('tooltip' option in the command line + GUI element in the dialog). See `make plot`
- LIGAND.displayDockingMoves option to control display of montecarlo moves in Ligand Editor docking.

Version 3.7-2f Mar 13 2014

- "locked" objects can be copied
- removed '*' from object/mol popup menus

Version 3.7-2e

- fixed rendering of down bond for scaled down compounds
- fixed false interpretation of '$' in single quoted strings

- quotes are processed properly in dialog input elements
- JSON parser accepts array as a top element: Collection("['a','b']")
- fixed atom naming in llp residue according to PDB convention
- Interrupt button cancels `read` from url
- `grob` clipping works with wire and dot representation.
- Fixed `read pdb` with 'non-standard' hydrogen names
- Optimized speed of the `write map` command
- Fixed inverted stereo in problem after "Extract ligand" in some cases.
- Mouse wheel in the workspace scrolls again.
- Fixed display of the variables in the Ligand Editor: Display Strain
- Fixed bug in copy/paste of single table column with empty rows.
- HTML tags from column format are not exported into SDF file

Version 3.7-2d

- Added docking hitlist filtering macro to get N top scored poses. scanFilterHitlist. (The option is also available in the GUI "Make hit list" dialog)
- fixed bug in `Formula` for structures with no hydrogen.
- Removed unnecessary ligand surface rebuilds in the Ligand Editor
- Fixed bug in `display slide` when skin representation wasn't preserved.
- `Index( X, X atom map` function is added to find atom number correspondence between two chemicals
- fixed nested index access to `collection`
- Fixed crash in `make map potential`
- Unix and Mac: "Save project As" works if path contains spaces.
- Windows only: -R option sets stdin/stdout to binary mode which makes possible to use pipe-able scripts in KNIME nodes
- fixed numerical instability in `minimize`
- fixed bug in `make reaction`
- `_confGen` and `_chemSuper` can be used in unix pipes and KNIME nodes
- `Min`/`Max functions work with datearrays
- Fixed bug recently introduced in `IcmSequence` function

Version 3.7-2c

Major new Features:

- New linker and core replacement tool in the 3D ligand editor
- Easy way to write pipe-able scripts (see $ICMHOME/molpipe/*.icm).
- Easy way to add parallelism to unix/mac ICM scripts: `fork` with `pipe` option ($ICMHOME\molpipe\*.icm)
- copy/paste and drag'n'drop between icm sessions (images, molecular objects, meshes, tables)

New Features:

- Added `Collection(` tlt.column format) to get the various column properties as a `collection` object.
- Added `Table( model chem [inverse] )` function.
- reading large file by chunks (`read file by chunk`) now supports zipped file.
- Score(*X_3Dn* [*X_3Dm*] [similarity|distance]) *M_nxm* apf_scores normalization of apf cross-scores.
- added option=s_filename argument for `make plot` command which allows one to generate plot images in the batch mode.
- added PDF/EPS format for saving plot as an image.
- `rename` of the object/mesh/map keeps slides and distances in consistent state.
- switching the receptor conformation in the ligand editor rebuild ligand and pocket surface if they are present
- `Nof( fork )` is added to get the number of available processors
- *pipe* options is added for `fork` command
- a_SLIDE. selection is added. Returns list of objects used in slides, also delete all compress
- `learn` remembers columns calculated with `add column function` to automatically recalculated in 'predict and `Predict`
- _chemSuper supports input from pipe.
- Slides now remember the conformation from the object's `stack`
- Added `stack` option for `read mol` command.
- `read mol` can read directly from `chemical array`
- `Score(` as_ as_ field) and `Score(` chem_array field) are added for APF clustering

- Added *stack* option for `read mol` command to read SD file with multiple conformations as an object with a `stack`.
- Added export alignment to PDF or EPS. (Right click "Save/Export To Image/Save Image")
- Added support of the APF ligand based project in the 3D ligand editor.
- `Srmsd(` .. .. [weight|chemical] matrix) returns sarray of static RMSDs or, with option `weight`, superposition errors computed according to the `TOOLS.superimposeMaxDeviation` (consistent with r3_out of `superimpose minimize`)
- `Volume(` *R6* box)
- Table(residue) now returns residues from the `icm.res` and usr.res libraries. New residues added for CME, CSD, CSO, HYP, KCX, LLP, PTR, SEP, TPO, TYS, and CIR modified amino acids.
- read binary pdb *s_4letter_code* to read .icb files in pdb-style subdirectories according to the `s_xpdbDir` root (also extended to allow `http` , e.g. `s_xpdbDir` = "http://xablab.ucsd.edu/xpdb/"
- `term ts` , `TOOLS.tsShape` and TOOLS.tsShapeData to allow soft penalties for atoms moving outside certain shapes (sphere, box).
- String(*s* [16|32] key) (hash function generating 8-character long keys) fixed; String(*s* hash) added to generate stronger keys of length 26 or 32
- added coloring for cluster nodes (heat maps)
- added `Index compare` function to compare different arrays
- `parray` supported as `collection` elements
- plot rendering speed is optimized significantly.
- added 'add exact' combination for `make grob map` command to allow absolute increment
- `delete` command works directly with index expression in `collection` (delete c["aaa","bbb"])
- individual atom ball&stick radius.
- added ability to read large portions of data by chunks (see: `read file by chunk`)
- added ND word. (for non-defined)
- Added HTTP.cookies `collection` to allow easier control over http cookies.
- GUI access to molecular named fields (see `set field`)
- added image rendering for 2D labels

Bug fixes:

- fixed bond removal in cyclopeptides in `convert` command.
- fixed crash in `build tautomer` command
- Fixed bug in map contouring with large buffer size values.
- Fixed bug in skin rendering in `display stack`
- Fixed bug in readUniprotWeb macro when l_references is 'yes'
- Fixed bug in "Filter Graphical Selection" dialog
- Fixed bug in S_ ~ s_ and S_ == s_
- Fixed rare crash in mc
- fixed bug in cis/trans smiles generation (when cis or trans bond is a ring closure bond)

Version 3.7-2b

- Fixed permission issue in Molcart on MySQL 5.5
- added output= argument for all `show` commands which allows one to grab the output for further processing.
- Improved rendering speed of xstick representation
- Fixed cis/trans depiction for R-group substituents in `enumerate library`
- Fixed memory few memory leaks when working with tables in macros.
- Fixed chemical formula generation for isotopes.
- copy/paste multiple cells from Excel into ICM spreadsheet.
- Fixed crash "Changing the final row in a filtered column crashes ICM" bugzilla: #697
- Fixed crash in pasting a column in the table with plot.
- XML parsing functionality. `read xml`, `Collection`, `xml drugbank example`
- fixed incorrect cis/trans bond interpretation in `smiles`. ((\R)C=C R/C=C)
- new attachment point rendering style
- fixed parsing bug in cell hyper links (`set format`)
- fixed "interrupted system call" error in read http.
- chiral centers with an R-group attached are correctly treated (stereo is preserved by parity) in `make reaction`.
- `smiles` property for `sarray` columns in a `table`. Toggles on-the-fly 2D chemical depiction.
- Fixed bug with non-preserving center of rotation in slides after clipping planes were moved. (also fixed in activeICM 1.1-5)
- Multi-receptor support in the 3D editor
- `make reaction` allows arbitrary R-group numbering
- multiple drag-able resize-able images in `set background image` command

- added 'Unlock ICB' function which clears read-only or password protection from ICB without reopening it.
- fixed few bugs in with password protected ICB files
- removing salts with `modify chem delete salt`
- preview for ICB files (Windows browser + ActiveICM)

Version 3.7-2a

- single quote string constant support.
- reader.icm script
- Logical(r|i|s)
- GRAPHICS.cpkClipCaps preference (1,2,3)
- SolveCubic(a,b,c,d | R_3|4 [all]) R, SolveQuadratic(a,b,c | R2|3)
- fixed overlapping shaded and bordered boxes in alignment view. All view preferences now stored in ICB.
- WebKit integration is added
- other types of occlusion shading added (TOOLS.occlusionColorStyle = "dark outside" or "light outside")
- fixed bug with opposite rotation of *connected* molecules/objects or grobs
- delete sequence nucleotide|protein|peptide
- fixed rdpdb for consecutive atoms with identical coordinates, e.g. 1a69, ds skin molecule,
- Nof(bond [error]) Nof(selftether [error]), Select(selftether [error]), Select(bond [error])
- `selftether` treatment: `set selftether` , delete selftether , Srmsd(*as* selftether), minimize "ts" , replaced tether= by selftether= in `minimize` `montecarlo` ; convert to set `selftether`
- improved `convertObject` (protonation of a_/U, minimize "ts" for the convert deviants)
- added *output* option for `Rmsd` and `Srmsd` function to store individual deviations into `R_2out`
- `grouping by table column` works with ND values correctly in aggregation operations.
- write gamess to write correct Z-matrix, memory limits.
- TOOLS.membrane, update icm.hdt, surfaceMethod="membrane" supported in Area(energy) and energy evaluations.
- bug with residue names for residue names starting from a leading blank after reading some pdbs fixed. (previously was breaking 1ytw,2i42,3f9a,2i4e,1kpe)
- CubicRoot(|)
- bug with crashes in find database fixed
- improving find database results (prioritize SC for HUMAN, ECOLI, BOVIN etc. in case of identical scores)
- a_J selection to filter short peptides.
- Name(sequences) returns an empty array (not error) when there are no sequences
- `set column format` supports internal ICM links.
- `exclude` and `number` options for `learn atom` command.
- `set atom named fields` is added
- anti-aliased and scalable fonts (atom/residue labels,etc) in 3D for windows and MAC
- Pattern(*rs*), selections a_/B*barcode_like_A12L2L* and a_/Q*barcode*
- translation to a destination point for a molecular and object selections (see `translate`)
- new 2D compound rendering option: color rectangles on hetero atoms instead of atom labels. See `set property chemical view`
- fixed crash in `Askg`
- new `Iarray`(a_// topology) function
- new `align sequence` command (align number ..)
- new `Select_by_sequence` function
- `find molecule sstructure drestraint` *ms1_inIcmObj ms2_inTheSameIcmObj* added to set drestraints by chemical similarity
- `delete link alignment` ; `link` *ms_* rewritten, many options added. autoLink action improved.
- `join` by structure column ignores hydrogens (3D can now be joined with 2D structures). `stereo off` option is added to ignore chirality.
- copy/paste and drag'n'drop between icm sessions (images, molecular objects, meshes)
- `build tautomer` and `set tautomer` can be applied to HIS residues
- ND support for individual real values
- large 64 bit integer support for `integer` variables
- `Grob`("cylinder", r_ra r_H)
- arbitrary length atom labels and ball radius (`set-label-atom and `set atom ball` command)
- search functionality in preference dialog.
- `restore preference` command is added (allows one to set system preferences and variables to default values)
- `date type` is introduced
- new annotation style for alignments

- Charge/protonation state prediction using pKa model
- pKa predictions for bases and acids (`set-charge-auto command)
- GUI parser patched and icm.gui cleaned from in-line argument declarations to allow natural command syntax (the percent symbol, like %i_out , is no long needed) in dialog descriptions
- option to avoid sampling His, Gln, Asn, etc. and hydrogens upon convertObject added (allows one to keep atoms as is)
- concurrent multiple level contouring (with multiple colors) of distant density added
- `read pdb` improved to recognize unusual amino acids
- mmff treatment of two nitrogens in a fused aromatic ring improved
- `set field` can be used with `alignments`
- `set color alignment` is added
- Header(grob)
- `make distance append` behavior corrected (tool first order). Option `make distance append` *P_atompairs* .. added.
- `superimpose` *P_atompairs os_movable* command added
- `set tether` *P_atompairs os_movable* command added
- `Area(` *grob* `error`) returns the percentage of unclosed area (to detect surfaces that are not fully closed)
- `Volume(` *grob*) returns the percentage of unclosed area in `r_2out`
- copyMol, moveMol and jumping molecules
- not properly recognized chemical templates in 2D chem drawing
- unusual amino acids are not recognized (2jge)
- mmff treatment of two nitrogens in a fused aromatic ring problematic
- adding ligand (without receptor) to the table causes icm/C5H5N> processLigandSave Error> [2191] index 1 of array out of range [1:0]

Brief history of ICM

ICM author's heads "*in italic*"

Ruben Max

The first lines of ICM were born in 1985 out of a desire to design a fast yet general framework for predicting the structure of complex biological macromolecules and their complexes. I formulated a set of requirements for a program for molecular mechanics in a full set of internal coordinates, and started working on the internal coordinate algorithms and the Fortran code of the first program blocks. By 1991 the batch parameter files were replaced by a command language and an interactive shell that looked quite similar to the current version of ICM; the molecules started to follow commands and sample the energy minima.

Max Totrov and I extended or rewrote most parts of ICM from 1991 to 1994. By 1993 several people (Alexey Mazur, Mikhail Petukhov, and Dmitry Kuznetsov) had also contributed to the fortran version of ICM, however their contributions did not survive in the current version of the program. Alexey pursued the development of molecular dynamics in internal coordinates which was first formulated and tested in a series of papers in 1989 and, later, branched out of ICM.

The all-C version of ICM emerged in 1994 as a result of a full rewrite. Some features were lost, but more were gained. **Serge Batalov** joined the development of the program in the fall of 1994, about the time Molsoft was founded. Another contributor to the code was Levon Budagyan.

Eugene is writing the graphics user interface, chemical functions and pretty much anything else. The three of us work together to keep ICM strong, clean, healthy and alive.

Eugene

ICM distribution and support

ICM is being developed, distributed and supported by Molsoft, LLC.
If you have any problem with our programs, go to the Molsoft Support Center at

```
http://www.molsoft.com/help.html
```

or contact Molsoft via e-mail:

```
support@molsoft.com
```

In the support center you can easily post a problem or make a suggestion and monitor its progress. Please indicate the platform, the version of the program, and do not forget all the necessary files to reproduce it. Some of the commands or functions described in this manual belong to specific modules and are not available in the ICM-main program.

Note that the Molsoft website contains online documentation only for the latest version of the program.

What can you do with ICM? (a program overview)

Let us go through the short overview of the ICM application areas.

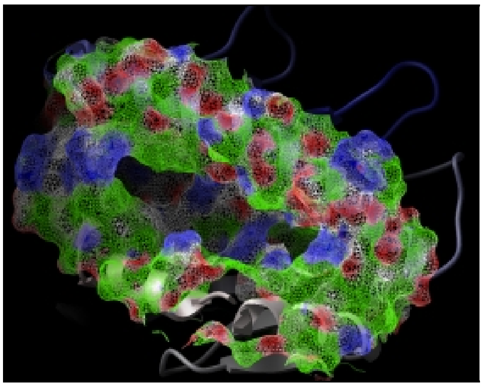
Graphics

ICM and ICM-derived plugins provide a viewing environment for large a small molecules and general three-dimensional objects with or without textures. Various types of enhancements including stereo, anti-aliasing, graphical layers, on-the-fly generation of shadows, occlusion shading, custom backgrounds, depth cueing and simple rotation, translation, zooming, clipping, picking, continuous movements, separate

Versatile surface and structure views to elucidate protein function

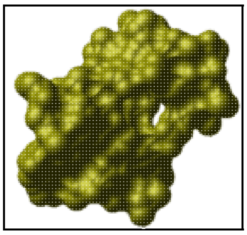
The views include

- binding and active site surfaces with mapped properties
- automatic identification and views of cavities and open binding pockets
- electrostatic surfaces



Analytical molecular surface (skin)

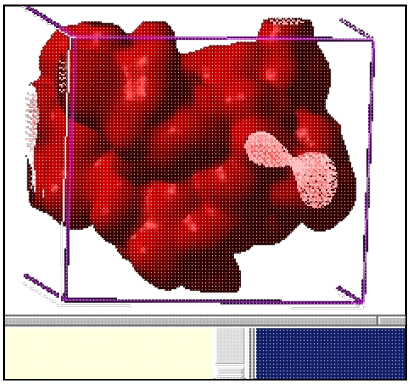
The contour-buildup algorithm calculates the smooth and accurate analytical molecular surface in seconds. This surface can be saved as a geometrical object, saved as a vectorized postscript file.



The skin is used in the REBEL algorithm to solve the Poisson equation, as well as in the molecular surface analysis routines (e.g. a projection of physical properties on the receptor surface).
Also ICM can build and draw a solvent-accessible surface (see `surface`) and

* a Gaussian molecular density which can be contoured at different levels and to generate different smooth molecular envelopes and enclosed pockets and cavities:

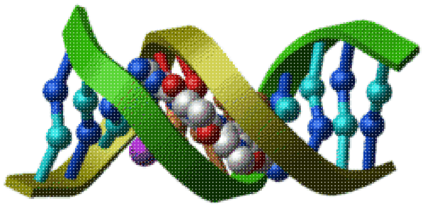
```
build string "HEK" ; display a_ xstick # tripeptide
make map potential Box( a_ 3.)
make grob m_atoms exact 0.5 solid
display g_atoms smooth transparent
```



Schematic representations of DNA and RNA

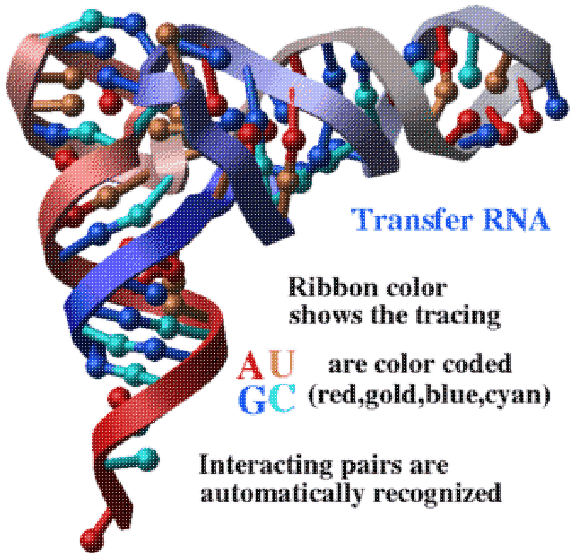
PDB entry: `101d`
ICM command:

```
nice "103d"
```



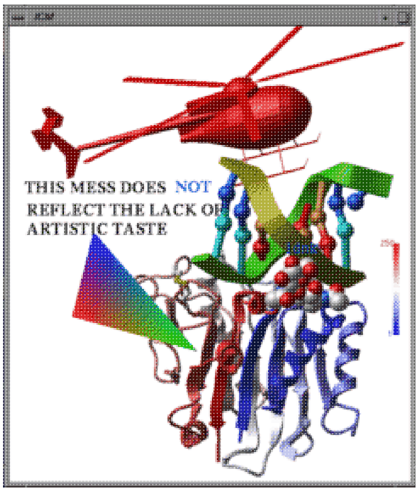
PDB entry: `4tna`
ICM commands:

```
nice "4tna"
color ribbon a_N/* Count(Nof(a_N/*))
```


Transfer RNA
Ribbon color shows the tracing
AU are color coded
GC (red,gold,blue,cyan)
Interacting pairs are automatically recognized

Complex combined representations

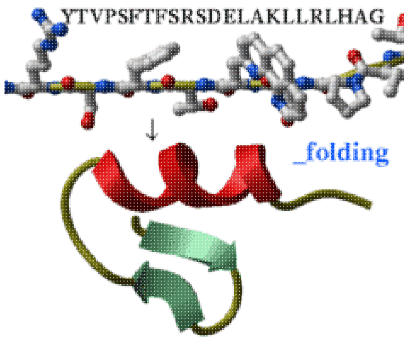
Simplified molecular representations are built automatically (e.g. the protein-dna complex is shown with one command: `nice "1dnk"`). You can combine different types of molecular representations with solid or wire geometrical objects.



Molecular representations include wire models, ball-and-stick models, ribbons, space filling models, and skin representation.

Simulations

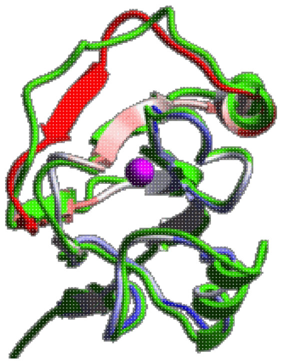
Prediction of peptide structure from sequence



Take a peptide sequence and predict its three-dimensional structure. Of course, success is not guaranteed, especially if the peptide is longer than about 25 residues but some preliminary tests are encouraging.

You will also get a trajectory file of your peptide folding up which can be interactively watched. Just type the peptide sequence in the `_folding` file and go ahead.

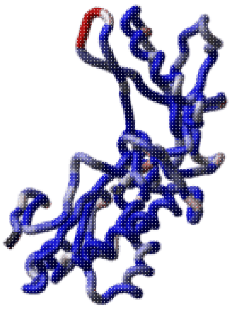
High quality models by homology



ICM has an excellent record in building accurate models by homology. The procedure will build the framework, shake up the side-chains and loops by global energy optimization. You can also color the model by local reliability to identify the potentially wrong parts of the model.

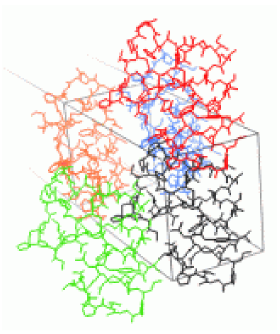
ICM also offers a fast and completely automated method to build a model by homology and extract the best fitting loops from a database of all known loops (see `build model` and `montecarlo fast`). It just takes a few seconds to build a complete model by homology with loops.

Loop modeling and protein design



ICM was used to design two new 7 residue loops and in both cases the designs were successful. Moreover, the predicted conformations turned out to be exactly right (accuracy of 0.5A) after the crystallographic structures of the designed proteins were determined in Rik Wierenga's lab. Use the `_loop` script to predict loop conformations and calcEnergyStrain to identify the strained parts of the design.

Crystallographic symmetry



ICM has a full set of commands and functions to generate symmetry related molecules and generate "biological units".

Docking two proteins

Docking two proteins reliably is still an unsolved problem. However, there has been a considerable progress. In some cases (e.g. beta lactamase and its protein inhibitor) the ICM docking procedure predicted the binding geometry correctly based only on the global energy optimization. ICM will generate a number of possible solutions using both the explicit atom model of the receptor and the receptor grid potential and refine them by explicit global optimization of the surface side-chains. Even though success is not guaranteed, the generated solutions can be useful, especially if any additional information about the binding is available.

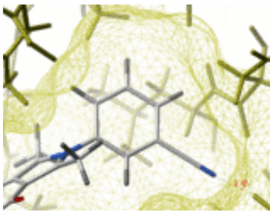
Finding pockets and docking a flexible ligand to a receptor

As demonstrated in several recent papers, short flexible peptides can be successfully docked *ab initio* to their receptors. This method is a blend of the peptide folding with the grid potentials representing the receptor. A similar method can be applied to any chemical. A chemical can be built from a 2D representation and optimized. The "druggable" pockets can be predicted with an algorithm based on the contiguous grid energy densities.

Scanning a database of flexible ligands

In virtual screening the flexible docking is applied to hundreds of thousands of individual ligands. This version of docking is fast and requires an accurate relative binding or ranking function to discriminate between the true ligands and hundreds of thousands of potential false positives. The ligand sampling and docking procedure is a combination of the genuine internal coordinate docking methodology with a sophisticated global optimization scheme.

Accurate and fast potentials and empirically adjusted scoring functions have led to an efficient virtual screening methodology in which ligands are fully and continuously flexible.



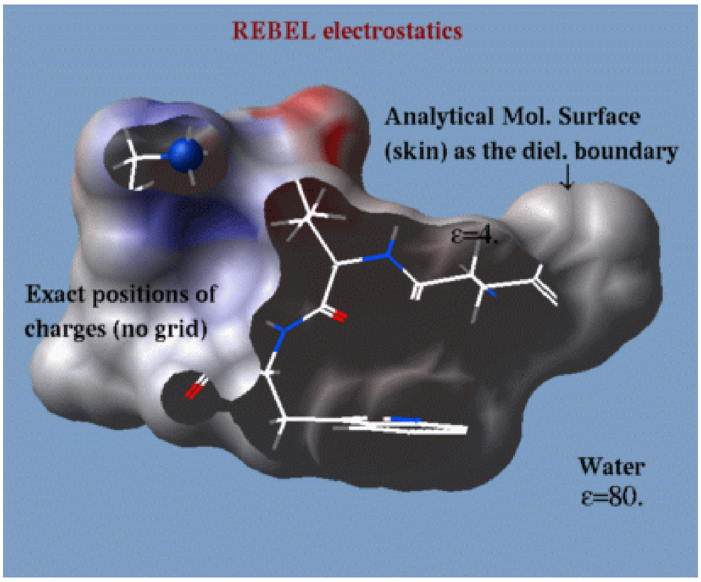
Interactive docking and focused library design

ICM allows one to draw a molecule directly in 3D with full undo/redo support and check its fit to a protein binding pocket. This environment is called a 3D ligand editor. The editor functionality is described in the User Manual.

Calculating electrostatic potential

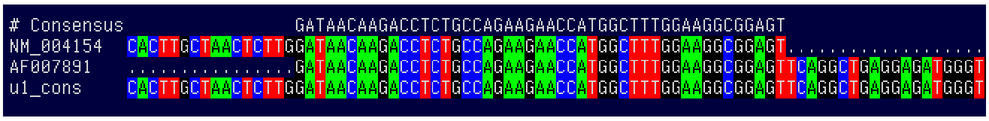
ICM incorporates a very fast and accurate boundary element solution of the Poisson equation to find the electrostatic free energy of a molecule in solution. This algorithm (abbreviated as REBEL) can be used dynamically during conformational search. The components of the electrostatic free energy are used to calculate the binding energy and evaluate the transfer energy between water and organic solvents.
ICM uses generalized Born approximation to calculate the electrostatic solvation energy and its gradient dynamically during local and global conformational searches.

The electrostatic potential can be projected on a molecular surface for the identification of possible binding sites.



Sequence analysis

Genomics



```
# Consensus                GATAACAAGACCTCTGCCAGAAGAACCATGGCTTTGGAAGGCGGAGT
NM_004154   CACTTGCTAACTCTTGGATAACAAGACCTCTGCCAGAAGAACCATGGCTTTGGAAGGCGGAGT...................
AF007891    ................GATAACAAGACCTCTGCCAGAAGAACCATGGCTTTGGAAGGCGGAGTTCAGGCTGAGGAGATGGGT
u1_cons     CACTTGCTAACTCTTGGATAACAAGACCTCTGCCAGAAGAACCATGGCTTTGGAAGGCGGAGTTCAGGCTGAGGAGATGGGT
```

Handling gigabytes of genomic sequence, fast cross-comparison of millions of sequences was another challenge solved in the ICM program. ICM can identify a unique subset of millions of sequences, assemble sequences from Unigene clusters into alignments (`SIM4` program is used a part of the procedure).

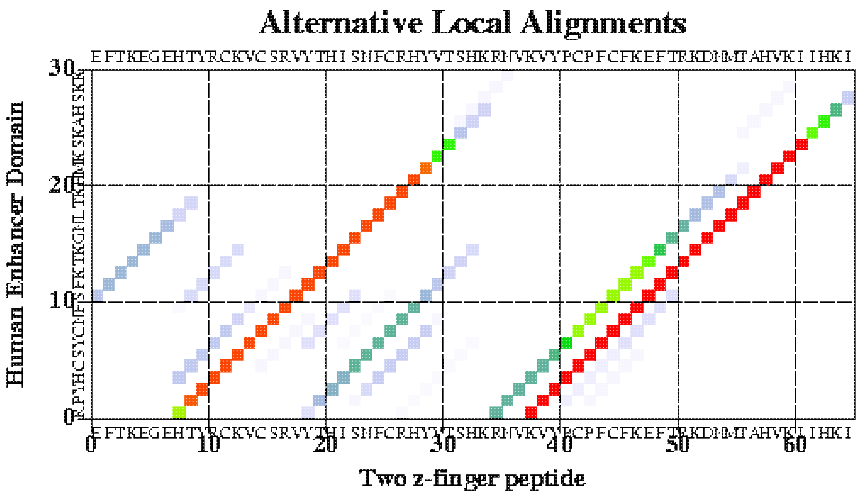
Similarity dotplot: alternative alignments and repetitive subdomains

It looks like this:
Using the `plotSeqDotMatrix` macro:

```
 read sequence s_icmhome + "zincFing.seq"
 plotSeqDotMatrix 2drp_d 3znf_m \
 "Two z-finger peptide" "Human Enhancer Domain" 5 20
```

(if the macro complains about `s_psViewer` , set it in Preferences/Directories , or reassign, e.g. `s_psViewer = "display" )`

Here the color shows the local significance of the alignment. You can change the method to calculate probability, color scheme and residue comparison matrices and calculate it interactively or in batch.



Pairwise sequence alignment and its significance

Make a pairwise sequence alignment and evaluate the probability that the two aligned sequences share the same structural fold. The alignment is performed with the Needleman and Wunsch algorithm modified to allow zero gap-end penalties (so called `ZEGA alignment`). The ZEGA probability is a more sensitive indicator of structural significance than the BLAST P-value. The structural statistics was derived by `Abagyan and Batalov, 1997`:

```
 read sequence s_icmhome + "sh3.seq"
 show Align(Fyn Spec)    # the probability will be shown
```

You can change residue comparison matrices, gap penalties and do many alignments in batch.

The ICM alignment functions and commands are summarized in the `alignment` section.

Multiple sequence alignment

Read any number of sequences in fasta or swissprot formats and automatically align the sequences, interactively or in a batch. It will look like this:

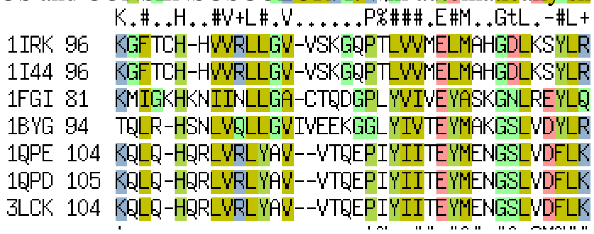
```
# Consensus ...#.^.YD%..+~..-#~# K~-.#~##.~~..~WW.#.    ~~.~G%#P.
Fyn      ----VTLFVALYDYEARTEDDLSFHKGEKFQILNSSEGDWWEARSLTTGETGYIPS
Spec     DETGKELVLALYDYQEKSPREVTMKKGDILTLLNSTNKDWWKVE--VNDRQGFVP-
```

```
Eps8    KTQPKKYAKSKYDFVARNSSELSM-KDDVLELILDDRRQWWKVR---NSGDGFVPN

# nID 7 Lmin 56 ID 11.5 %
#MATGAP gonnet 2.4 0.15
```

ICM commands:

```
 read sequence s_icmhome + "sh3.seq"
 group sequences sh3
 align sh3
 show sh3
```

The gui version of ICM also has a multiple alignment viewer with dynamic coloring according to conservation tables CONSENSUS and CONSENSUSCOLOR. It will automatically show secondary structure and other features.



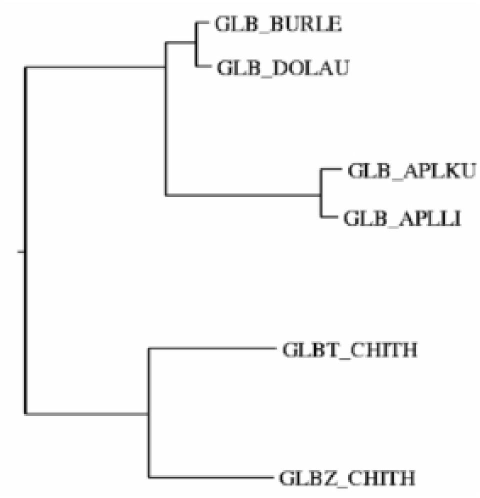
```
          K.#..H..#V+L#.V......P%###.E#M..GtL.-#L+
1IRK 96   KGFTCH-HVVRLLGV-VSKGQPTLVVMELMAHGDLKSYLR
1I44 96   KGFTCH-HVVRLLGV-VSKGQPTLVVMELMAHGDLKSYLR
1FGI 81   KMIGKHKNIINLLGA-CTQDGPLYVIVEYASKGNLREYLQ
1BYG 94   TQLR-HSNLVQLLGVIVEEKGGLYIVTEYMAKGSLVDYLR
1QPE 104  KQLQ-HQRLVRLYAV--VTQEPIYIITEYMENGSLVDFLK
1QPD 105  KQLQ-HQRLVRLYAV--VTQEPIYIITEYMENGSLVDFLK
3LCK 104  KQLQ-HQRLVRLYAV--VTQEPIYIITEYMENGSLVDFLK
```

The ICM alignment functions and commands are summarized in the `alignment` section.

Evolutionary trees, 2d and 3d sequence clustering

Relationships between sequences can be presented in three forms:

- as evolutionary trees (ICM uses the neighbor-joining method for tree construction);
- as 2D distribution of sequences using the two main principal axes (use plot2Dseq macro);
- as 3D distribution. This can be analyzed in stereo using controls of molecular graphics (use ds3D macro: `ds3D Distance(alig) Name(alig)`).



Sensitive Sequence Similarity Search, ZEGA

Search your sequence (interactively or in batch) through any database and generate a list of possible homologs which are sorted and evaluated by probability of structural significance. The `ZEGA` alignment (full dynamic programming with zero end gaps) is used for each comparison and an empirical probability function described in `JMB,1997` is used to assign a P-value to each hit. This search may give you more homologs that a BLAST search! The output may presented in a linked table form:

Table of hits

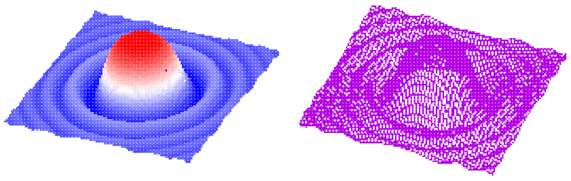
NA1	NA2	ID	SC	pP	DE
Fyn	`1nyf`_mNo	100.	62.81	20.94	fyn
...	*lines skipped*	...	...	...	...
Eps8	`1tud`_m17	21.	17.04	4.17	alpha-spectrin
Eps8	`1fyn`_a23	22.6	17.02	4.16	phosphotransferase fyn

Eps8	1efn_a25	22.	16.64	4.11	fyn tyrosine kinase
Eps8	1hsq_mNo	24.2	16.87	4.1	phospholipase c-gamma (sh3 domain)

3D plots of functions

Take a `matrix` and represent it in 3D in a variety of forms. View it in stereo, color, label, transform with the mouse. Example:

```
read matrix s_icmhome + "def"
make grob def solid color
display
```



Modules of ICM

ICM is distributed in the following packages:

- ICM-browser and ICM-browser-pro (distributed from a single package)
- ICM-chemist and ICM-chemist-pro (distributed from a single package)
- ICM-pro with options including bioinformatics, Poisson electrostatics, chemistry and cheminformatics, homology modeling, docking, virtual ligand screening (VLS).

ICM is distributed for the following three main platforms:

- Windows
- Linux and Unix
- Macintosh

There is a full ICM file compatibility between the platform. Also, the appearence of the GUI is identical. ICM command language contains around one hundred commands and one thousand functions operating on 20 different types of objects.

The modules have the following features: **ICM-main**

- shell for molecules, numbers, strings, vectors, matrices, tables, sequences, alignments, profiles, 3D maps, 3D graphical objects, 2D chemical tables/spreadsheets, images
- ICM-language and macros
- graphics, stereo
- imaging and vectorized postscript
- animation and movies
- mathematics, statistics, plotting
- presentation of the results in html format
- user-defined and automated interpretation of web links
- HTML-form-output interpretation
- pairwise and multiple sequence alignments, evolutionary trees, clustering
- secondary structure prediction and assignment, property profiles, pattern searching
- superpositions, structural alignment, Ramachandran plots
- protein quality check
- analytical molecular surface
- calculations of surface areas and volumes
- cavity analysis
- symmetry operations, access to 230 space groups
- database fragment search
- identification of common substructures in PDB
- read pdb, mol2, csd, build from sequence
- energy, solvation, MIMEL, side-chain entropies, soft van der Waals, tethers, distance and angular restraints
- local minimization
- ab initio peptide structure prediction by the Biased Probability Monte Carlo method
- loop simulations

- side-chain placement

ICM-REBEL (electrostatics)

- electrostatic free energy calculated by the boundary element method
- coloring molecular surface by electrostatic potential
- binding energy (electrostatic solvation component)
- maps of electrostatic potential and its isopotential contours

ICM-docking

- indexing of chemical databases in SD, mol2 and csd format
- searching and extracting from the indexed databases
- fast grid potentials
- scripts for flexible ligand docking
- scripts for protein-protein docking
- 2D (SMILES) to 3D conversion, type and charge assignment, mmff geometry optimization, low-energy rotamer generation
- refinement in full atom representation

ICM chemistry (also, see `here`)

- scripting access to the internal chemical spreadsheets and external chemical databases in MySQL (via `molcart`) and SQL lite.
- Various operations on chemical tables (see below)
- integration with the docking engine and interactive ligand editing

Operations on chemical tables:

- Calculate various properties and descriptors from 2D chemical table
- Standardize chemical structure (change ambiguous depictions of some functional groups to a standard form according to user defined tables)
- Build QSAR type prediction models from a chemical spreadsheet and apply those models to new chemical tables
- Convert Smiles to 2D and the opposite operation
- Generate 2D Depiction from a 3D or 0D chemical
- Convert a 2D chemical to 3D
- Generate 3D Conformers
- Generate Tautomers
- Generate Stereoisomers, assign and manipulate stereo centers
- Align/Color By 2D Scaffold
- Cluster chemicals by either fingerprint similarity or external distance matrices
- Compare two chemical sets for common elements
- Sort a table and select duplicate rows in a table
- Create/Modify Markush objects
- Enumerate a combinatorial chemical library from scaffold and R-groups
- R-Group Decomposition of a chemical spreadsheet
- Enumerate a chemical library by reaction(s) and reactants
- Various forms of multiple chemical superposition (both 2D and 3D)

ICM-bioinformatics

- fast comparison and redundancy removal of millions of genomic or protein sequences
- multiple EST clustering, alignment and consensus derivation
- database indexing and manipulations
- functions to evaluate sequence-structure similarity
- scripts to recognize remote similarities in the protein sequence and PDB databases
- search a pattern through a database
- searching profiles and patterns from the Prosite database through a sequence
- HTML representation of the search results with interpretation of links
- interactive editor of sequence-structure alignment
- automated building of models by homology with loop sampling and side-chain placement (fast homology model building combined with the database loop search is a separate module which is ICM Homology).

ICM-Homology

- sequence-structure alignment (threading)
- ultra-fast automated homology model building with a database loop search
- loop modeling and refinement, side-chain placement
- surface analysis

As a method for structure prediction, ICM offers a new efficient way of global energy optimization and versatile modeling operations on arbitrarily fixed multimolecular systems. It is aimed at predicting large structural rearrangements in biopolymers. The ICM-method uses a generalized description of biomolecular structures in which bond lengths, bond angles, torsion and phase angles are considered as independent variables. Any subset of those variables can be fixed. Rigid bodies formed after exclusion of some variables (i.e. all bond lengths, bond angles and phase angles, or all the variables in a protein domain, etc.) can be treated efficiently in energy calculations, since no interactions within a rigid body are calculated. Analytical energy derivatives are calculated to allow fast local minimization. To allow large scale conformational sampling and powerful molecular manipulations ICM employs a family of new **global optimization** techniques such as: Biased Probability Monte Carlo (`Abagyan and Totrov, 1994`), pseudo-Brownian docking method (`Abagyan, Totrov and Kuznetsov, 1994`) and local deformation loop movements (`Abagyan and Mazur, 1989`).
A set of ECEPP/3 **energy** terms is complemented with the parameters for rare atoms and atom types, as well as the solvation energy terms, **electrostatic polarization energy** and side-chain **entropic** effects (`Abagyan and Totrov, 1994`), making the total calculated energy a more realistic approximation of the true free energy. The MMFF94 force field has also been implemented. Powerful molecular graphics, the ICM-command language, and a set of structure manipulation tools and penalty functions (such as multidimensional variable restraints, tethers, distance restraints) allow the user to address a wide variety of problems concerning biomolecular structures.

Notational conventions

The following notational and typographical conventions are used throughout the manual.

- **Bold.** Command names may appear in bold in syntax descriptions. (e.g. **montecarlo**). Type them as they appear in the text.
- `Typewriter` font is used for command words, examples and ICM-shell prompts. This `text` can also be copied into the shell.
- *Italic* font is used for command or function arguments which should be replaced with actual values. For example, if you see */whatever/your/ICM/directory/* and your ICM directory actually is `/usr/pub/icm` the latter is what you should actually type. Short prefixes shown in parentheses may be used to specify argument type: `integer` (i), `real` (r), `string` (s), `logical` (l), `preference` (p), `iarray` (I), `rarray` (R), `sarray` (S), `parray` (P), `date array` (e), `parray` of chemical molecules (X), `matrix` (M), `sequence` (seq), `profile` (prf), `alignment` (ali), `map` (m), `graphics object, or grob` (g), `structure factor` (sf), `atom selection` (as), `residue selection` (rs), `molecule selection` (ms), `object selection` (os), `variable selection`, e.g. a subset of torsion angles, (vs), and `table` (T). `chemical table` (X). These prefixes are also used to construct formal argument names for `macros`. For example, *I_Color* would mean an integer array with color information, or *s_ObjName* would mean a string variable or constant (e.g. `"crn"`) specifying the object name.
- Optional arguments appear in square brackets **[ ]**.
- Braces **{ }** are used for mutually exclusive groups or arguments. For example: `set charge` *as* { *r_Charge* | `add` *r_Increment* } means either `set charge` *as r_Charge* or `set charge` *as* `add` *r_Increment*
- The default values in ICM macros are shown in parenthesis and in typewriter font: `icmPocketFinder` *as_receptorMol r_threshold* (`3.`) *l_display* (`yes`)

Sometimes the *dimension* of an array is shown after the underscore symbol, e.g. *R_3xyz*, means that this is a 3-membered array.

Common abbreviations

In addition to the abbreviated ICM-shell-objects prefixes (see above), abbreviations may be used for energy terms, and some other frequently used words.

abbr.	description
as_	atom selection
ali	alignment

conf	conformation
cn	distance restraints
grad	gradient
ey	energy
hb	hydrogen bonds
ls	list
ms_	molecular selection
MC, mc	montecarlo
MB	Mouse Button
mn	maximal number of *items*
n	number of *items*
os_	object selection
re, res	residue
rs_	residue selection
rs	variable restraint
seq	sequence
to	torsion
tz	tether
ty	type
va, var	variable internal coordinate in a molecule (torsions, phase angles, planar angles, bond lengths).
vw	van der Waals
wt	weight
X_	array of 0D,2D or 3D chemicals

It is convenient to declare these abbreviations as aliases to the corresponding full words in the `_startup` file for fast typing. For example:

```
 ls seq
```

instead of

```
 list sequence
```

Getting started

Start the `GUI` (Graphics User Interface) version of ICM by typing `icm -g` or `icm -G` and hitting `RETURN`. This executable will look the `$ICMHOME` shell variable. The commands of the GUI menu will be taken from `$ICMHOME/icm.gui` file. Feel free to change it. The GUI is meant to be self-explanatory. In this manual we will mostly focus on the shell commands and function, since in many cases the GUI gives you only limited subset of possibilities.

ICM-shell

ICM-shell is a basic interface between a user and the ICM-program. The shell can be used from the GUI version or directly. This is a powerful and flexible environment for a multitude of versatile tasks ranging from mathematics and statistics to very specialized molecular modeling tasks.
Start ICM by typing:

```
 icm
```

Make sure that your .cshrc login file contains

```
 setenv ICMHOME  /whatever/your/ICM/directory/is/
```

Do not forget the slash at the end. It is also useful to add your $ICMHOME directory to your $path since there are some ICM related shell scripts and utilities which you may want to access.
You will see the ICM-prompt inviting you to type a command. The first thing to know is **how to get help**. You may just type **help** and use **/** *whatever* to find what you want, or use **help commands** or **help functions** to find out about the syntax. Now type:

```
 aa=2.4
```

You have just created a new ICM-shell variable `aa` and assigned a value of 2.4 to it. You can create a variable with a name which is not already in use in the ICM-shell, does not contain space or delimiters like ".","," and starts from a letter (e.g. 1aag is an illegal name, except for sequences). Let us go on:

```
 bb=2.*aa
```

Now you have created another ICM-shell variable `bb` and its value is probably 4.8. Find it out by typing:

```
 print " bb=", bb
```

or any of these commands:

```
 list "b*"
 list integers
 show bb
```

The next step would be to type a conditional expression like:

```
 if (bb != 4.8) print "something went wrong"
```

or something even more elaborate:

```
 if (bb != 4.8) then
   print "something went wrong"
 else
   print "It really works"
 endif
```

You can always start a for-loop such as:

```
 cc={"sushi","sashimi","negi maki","toro","period."}
 for i=1,Nof(cc)
# Nof returns the number of elements.
#  Index i runs from 1 to 5
   print "*** I just like to eat  ",cc[i]
 endfor
```

Notice that anything after a pound sign # in ICM scripts is a comment.
We have just played with a real variable `bb` and string array `cc` . They had their unique names and we could create, read, write, delete and rename them.
ICM-shell objects
Furthermore, the ICM-shell can handle many other different types too, namely, it may contain in its memory entities of 16 different types, such as

- `integer` , (e.g. `a=10, b= -3`)
- `real` , (e.g. `c = -3.14`)
- `string` , (e.g. `d = "ICM rules"`)
- `logical` , (e.g. `e = (2 > 43); f = yes`)
- `preference` , (i.e. fixed multiple choices, try `show wireStyle`)
- `iarray` , (i.e. integer arrays, `g={-2,3,-1}`)
- `rarray` , (i.e. real arrays, `h={ -2.3, 3.12, -1.}`)
- `sarray` , (i.e. string arrays, `i={"mek","yerku","erek"}`)
- `parray`, including array of 0D,2D or 3D chemicals, e.g. `chm = Chemical({"CC","CC(=O)O","C1CC1"})`
- `matrix` , (read from a disk file, e.g. `read matrix "def.mat"`)
- `sequence` , (i.e. amino acid or nucleotide sequences, e.g. `a=Sequence("ASDQWE")`)
- `alignment` , (i.e. pairwise or multiple sequence alignments, read from a file)
- `profile` , (i.e. protein sequence profiles)
- `map` , (i.e. density functions defined on the 3D grid)
- `grob` (abbreviation for GRaphic OBject, which is different from molecular graphics objects, and contains dots, lines and solid surfaces; it can be a contoured electron density, 3D plot, an arrow, etc)
- `atomic/molecular objects` and related selections of `atoms` (`a_//ca`,c,n) `residues` (`a_/2:15`) `molecules` (`a_1`.b,c/) `objects` (`a_1`,2.) and, finally ..,
- `table` , or spreadsheet. Several arrays are linked together in a table. Table can also have a header with some additional data fields. Tables are essentially simple databases which can be manipulated with, sorted and searched with ICM commands.

The more complicated objects, like arrays, sequences, alignments, maps etc., can be read from a disk file (e.g. `read sequence "a.seq"`) or created by an ICM command or function (e.g.

a=Sequence("ACFASDTRSEEDFFF") or make sequence a_1.1)
Atomic objects are usually specified by an atom, residue, molecule or object selection which are collectively referred to as selections.
All of the listed entities have their unique names in the ICM-shell and can be read, renamed (e.g. rename myFactors bbb), deleted (e.g. delete myFactors aaa), written to a file with a standard type-specific extension (e.g. write aaa "surf" will create file surf.gro , the extension type depends on the object), shown, often printed and displayed graphically.
A number of ICM-variables have reserved names and are used by the program. For example, the mncalls variable always describes the number of molecular energy evaluations during a minimization, s_pdbDir is the path to your pdb files, etc. You may customize some of those ICM-shell variables by redefining them in the system-wide _startup file, and $HOME/.icm/user_startup.icm file. The standard _startup file reads icm.ini file which contains many standard directory and parameter definitions, e.g.

```
 read all s_icmhome+"icm.ini" # initialize icm variables
```

Important: be careful when negative numbers appear in the command line. If not separated from the previous numeric argument by a **comma**, they will be interpreted by ICM-shell as an expression, i.e. the two arguments will simply be replaced by their difference. For example, the command

```
 display string "I like crambin" -0.9 -0.3
```

is **wrong**, a comma is needed, otherwise -0.9 -0.3 will be substituted by -1.2. This command will place the string in a point with screen coordinates X=-1.2 and Y=0.0 (the default), not in X=-0.9 and Y=-0.3 as might be expected. The safest way should be to use commas as separators in the argument list in the command line, like the following:

```
 display string "I like crambin" -0.9 , -0.3
```

is correct, the two arguments are separated by comma
Now you can use the mouse to rotate and translate molecules and strings. The left mouse button is associated with rotation, the middle mouse button is translation and the right mouse button clicks are used for drop down menus in GUI and labeling (double click is a residue label). A more detailed list of graphics controls is given below.
As far as the keyboard commands and prompting, try to use the arrow keys for invoking previous commands and TAB for prompting (e.g. atom **TAB**) to see the available commands and functions.

The first steps

Your first ICM commands may be the following:

```
 read pdb "1crn"   # check pdbDirStyle variable for PDB access
 display ribbon
```

or simply

```
 nice "1est"
```

You can also:

```
 read mol  s_icmhome + "ex_mol"  # or
 read mol2 s_icmhome + "ex_mol2" # or
```

The second way to create a molecular object is building the extended chain given the amino-acid sequence. The simplest way to build a short peptide is to use the build string command. Type

```
 build string "nter ala his leu tyr cooh"   # or
 build string "AHLY;AGGAR" # to build two molecules
 build string "ra rg ru; ra rc ru" # to build two rna chains
```

In a more complex case create a file, say mymol.se, .se being the standard extension for the object sequence files. The file should contain the names of molecules (field ml) and their sequence (field se) and may look like this:

```
ml mol1
se nter ala gly his ser trp cooh
```

```
ml mol2
se hoh
ml mol3
se hoh
```

Type:

```
 build "mymol"
```

to build the object. Now you can display the three molecular objects you have just loaded, i.e. crambin, the two peptides. We will use the `cpk` and the `xstick` graphics representations.

```
 display a_2.           # a_2. means 'the second object'
 display cpk a_1./2:10  # a_.. means 'residues 2:10 of the first object'
 display xstick a_1./16:18
```

You can also replace residues with the `modify` command:

```
 modify a_2./his "tyr"
```

Let us clear the scene and start doing some more fun things:

```
 delete a_*. # a_*. selects all the objects
 build "mymol"
 display     # by default displays everything
 set vrestraint a_/*  # this command will increase the efficiency
 montecarlo
```

Of course, there is a more elaborate possible setup for a montecarlo run (see `_folding` script) and graphics should not be used for a real run. However, the above example is pretty much what you need to do to run the Biased Probability Monte Carlo Minimization to find the global minimum which models the solution structure of this peptide.
Now let us make a quick tour into multiple sequence alignments. First, get your sequence file (most formats will be accepted). The simplest default file format (then you do not need format type specs like: msf, pir, etc) is the fasta format (angular bracket and sequence name followed by the sequence)

```
> seq1
ASDFREWWDYIEQ
> seq2
SDRTYIEQWWDCVN
```

There are some example multiple sequence files in the ICM-directory. Let us do the following:

```
 read sequences s_icmhome+"sh3"   # example sh3.seq file
 group sequence "*" sh3ali
 show sequences alignments
 align sh3ali                     # redo the multiple sequence alignment
 unix gs sh3ali.eps               # gs is a PostScript previewer
 show Align(Fyn, Eps8)            # make a pairwise alignment
```

If you want to go directly to more elaborate sessions and scripts, or have a "How can I ..." question, you may hop to the `User's Guide` section.

ICM Scripting Tutorials

If you are interested in learning more about the ICM command line language please download the Scripting Tutorials (the instructions are below). The tutorials contain a comprehensive guide to ICM scripting including a guide to the language, best practices, and worked examples. The interactive hyperlinked text in the icb files help you learn ICM scripting efficiently. Tutorials were prepared by Eugene Raush (Principal Software Developer, MolSoft LLC).

Instructions

- Please `download` (right click and Save Link As) the turorials.
- Unzip the file.
- Save the unzipped folder to a directory.
- Open the ICM graphical user interface and set the working directory to the location where you saved the files. To do this go to **Tools** menu and choose the option **"Change Working Directory"** and then browse for the directory.

- Go to File/Open and open one of the .icb files listed below and follow the links.
- Follow the html text in the icb files and click on the interactive links.
- Expand the size of the command line window so you can see the commands and the output.

Guide to the Tutorials

Scripting_Basics.icb

ICM Scripting Language Basics - Topics include:

- ICM Command Line
- Basic Data Types
- Control Structures
- Commands and Functions

Scripting_Workshop_MolObjects.icb

Molecular Objects - Topics include:

- Selections
- Internal Variable Selections
- Sequences
- Alignments
- Grobs

Scripting_Workshop_ICM_Scripts.icb

ICM Scripts - Topics include:

- Command line arguments
- Working with large SD files and piping
- SQL interface to relational databases
- Macros

Scripting_Workshop_Tables.icb

Working with Tables and Chemical Spreadsheets - Topics include:

- Tables introduction
- Table creation
- Collection(hash table)
- Deletion of columns and rows
- Filtering
- Columns with formulas
- Assigning custom actions to the table cells
- Grouping
- Plotting
- Chemical structures in tables
- Substructure and Similarity Search
- Annotate by matching fragments
- Find/Replace chemical groups
- Perform standardisation
- Clustering Trees
- Chemical objects vs 3D molecular objects

RegExpr.icb

Regular Expressions for Text Processing - Topics include:

- Simple expressions
- Repetitions and back-references
- Useful shortcuts
- Common tasks in the text processing
- Practical example: Conversion of DrugBank text format to SDF

Scripting_Workshop_GUI_Programming.icb

Creating your own GUI elements - Topics include:

- Dialog definition syntax
- Layout
- Referencing to the input values from the ICM command
- Ways to add a dialog to the interface
- Adding a link in the html page
- New top menu item
- Adding a button
- Askg() function

Scripting_Workshop_ActiveICM.icb

ActiveICM enables you to display fully interactive 3D objects in PowerPoint and Web. - Topics include:

- Client side
- GET and POST methods
- Server side

Reference Guide

ICM command line options

Option	Description
-a *arg_string*	initialize `s_icmargs` string with *arg_string*
-b	inhibit Buffered output
-c	clean: do not save _seslog
-e 'commands'	execute semi-colon separated icm commands and quit
-g [menuFile]	start GUI menu bar, using menu file [default=icm.gui]
-G [menuFile]	start GUI menu and keep the original terminal window
-i	Input Icb file from the stdin pIpe
-n	do Not execute _startup file
-ng	no-gui: open in shell only, do start GUI/graphics after reading the file
-R[] # Redirect standard text/info output into stderr to enable piping binary content	
-s	Silent mode (l_warn=no l_commands=no l_info=no l_confirm=no)
-p	set path for ICMHOME, e.g. -p/opt/icm/
-w	web cgi mode: combination of -p and -s, e.g. -w/opt/icm/
-d(or -display) address	sets/redirects the X display (default is $DISPLAY)
-24	enforce high quality 24-bit image mode at the expense of double buffering
-B[max_beeps]	no more than max_beeps on errors (default=300)
-X	report the computer identification number for a node-locked license GUI options:
-style={motif windows platinum cde}	sets the GUI style
-session=session	restores the earlier session
-geometry WxH+X+Y	sets the client geometry of the main widget, e.g. -geometry 200x200+150+700
-fn or -font font	defines the GUI font
-bg or -background color	sets the default background color
-fg or -foreground color	sets the default foreground color
-btn or -button color	sets the default button color
-name name	sets the GUI name
-title title	sets the title (caption)
-visual TrueColor	forces to use a TrueColor visual on an 8-bit display
-ncols count	limits the number of colors on a 8-bit display
-cmap	causes to install a private color map on an 8-bit display

Examples:

```
  icm -g                     # run in the interactive gui mode
  icm -g -p/home/yoda/icmd/  # temp path to ICMHOME
  icm 1crn.pdb 2ins.pdb      # opened these two files
  icm script.icm > script.out # runb
  icm -G  # keep the terminal window separate
  icm -a "4.2 3" # the string copied into s_icmargs variable
  icm -s  # no startup, just ICM in a quiet mode
  icm -24 # if you intend to save high-resolution images
  icm <<EOF
#command1     # ICM reads from standard input
#command2
EOF
  "print 123\nprint 432" | icm -s
```

Command line editing

(cursor is in the text window).

Operation	Shortcut Key
command word completion/prompting	**TAB**
up-history	**UP** arrow
down-history	**DOWN** arrow
forward-char	**RIGHT** arrow
backward-char	**LEFT** arrow
beginning-of-line	**CTRL**+A
delete-char	**CTRL**+D
end-of-line	**CTRL**+E
backward-delete-char	**Backspace** or **CTRL** F+H
kill-to-line-end	**CTRL**+K
insert-overstrike toggle	**CTRL**+O
paste	**CTRL**+P
delete/copy-whole-line	**CTRL**+U
delete/copy-word	**CTRL**+W
yank (identical to paste)	**CTRL**+Y

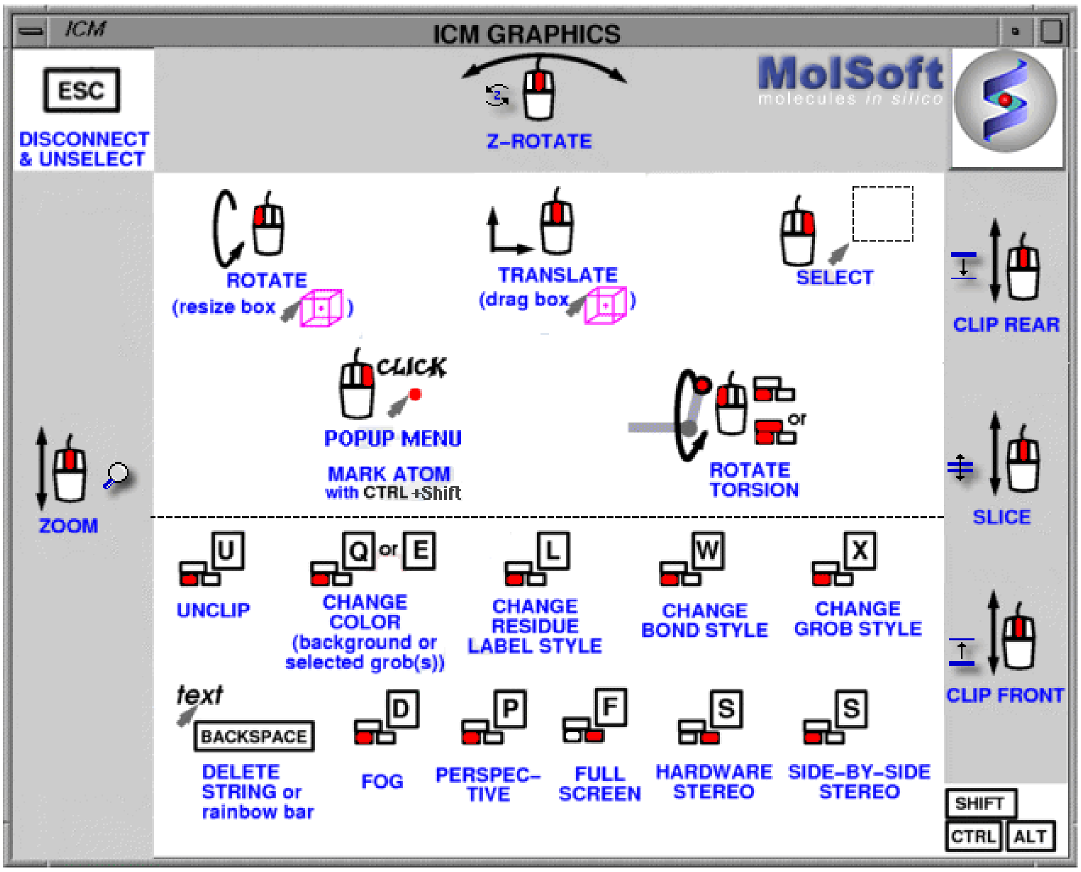
Use the **TAB** key when you do not know what to do or to **avoid unnecessary typing** as well as probable typos in long names. This prompting is very convenient and is consistent with the **tcsh** UNIX shell. It will not only prompt you for possible completions, but also prompt you for available files in the `read` command (hit TAB after the double quote mark) and available selection of items in `preference` .
Examples:

```
show Ic TAB      # completes function name IcmSequence()
read pdb "TAB    # gives you all local *.pdb *.brk files
read sequence "1a TAB  # lists 1a*.seq files
```

Graphics controls

The rough picture is simple: rotate with the *left* mouse button, translate, drag, crop, and zoom (drag along the left window margin) with the *middle* button, and select/pop with the *right* button. However these are only the defaults which can be customized.
The default shortcut keys are stored in the `icm.clr` file. Therefore the mapping of keys/mouse buttons to particular graphics operations is *flexible* and can be easily redefined. The `GUI` controls and the **popup** menu are additional to the older shortcut keys listed here. The following shortcut keys to speed up operations in the graphics window (see the quick graphics reference guide) are defined by default. If some of these definitions are not working, check your `icm.clr` file in the `$ICMHOME` directory and modify the key/mouse-operation mapping to your liking.

Quick graphics reference guide



It is necessary to have the cursor in the graphics window. For some operations you may need to move cursor in a specified area (e.g. left margin) of the window. (Note for Windows 95/Windows NT version's users: if you use a two-button mouse hold the left button and the SPACE key instead of the middle mouse button (see picture-prompt for two-button mouse). Some controls use only a margin on the screen (e.g. Bottom5 means the bottom 5% of the graphics viewing area).
Note: if your SGI hardware stereo does not work properly you may need to install IRIX6.4 patches 2448, 2771 and 2843.

OPERATION	DESCRIPTION	KEYS
	simple	LeftMB (MB stands for Mouse Button)
	continuous	**Shift**-Bottom5-LeftMB
ROTATE SHIFT key enforces global rotation	Z-axis clockwise	LeftMB at top margin (*or* **ALT** +Z)
	Z-axis counterclockwise	LeftMB at top margin *or* **CTRL** +Z
	individual torsion angle in ICM-object	**CTRL** (or **CTRL+SHIFT)** LeftMB on reference atom
TRANSLATE	XY-plane (dragging)	MiddleMB
SHIFT key enforces global translation	drag atom in non-ICM object	**CTRL** LeftMB at the dragged atom
GRAPHICS.resLabelDrag controls residue label dragging	Z-axis	MiddleMB at right margin

ZOOM	zoom in	MiddleMB at left margin or **SHIFT** MiddleMB up
	zoom out	MiddleMB at left margin or **SHIFT** MiddleMB down
CLIPPING PLANES	front plane	**CTRL** MiddleMB
	back plane	**ALT** MiddleMB or Right5-MiddleMB
	slice/slab (move both planes)	**CTRL**+ALT MiddleMB
	unclip	**CTRL**+U
LABELING	label atom or grob	RightMB-click
	label residue	double RightMB-click
	paste atom's/grob's name to command line	**CTRL-SHIFT** RightMB (or under Gui: RightMB on atom and release on 1st item)
	paste residue name to the command line	**CTRL** double-RightMB (GUI: RightMB on residue, popup menu and release on 1st item. Use the residue selection level, R)
	set 3D cursor to the residue (move with arrows)	**CTRL-SHIFT** double RightMB
CONNECT for independent movement of molecule(s) **SELECT GROB(S)** for changing size or color	disconnect/unselect everything	**ESC** or double RightMB-click, cursor in any empty area of the screen
	connect to molecule or grob	**CTRL**+ALT RightMB-click on atom or vertex
	connect to more molecule(s)/grob(s)	**CTRL**+ALT+SHIFT RightMB-click
	select/edit grob	double RightMB-click
	add new grob to a selection	**SHIFT** double LeftMB-click on grob
MODES	side-by-side stereo on/off	**CTRL**+S
	hardware stereo on/off	**ALT**+S
	full screen on/off	**CTRL**+F
	perspective view on/off	**CTRL**+P
	fog (`depth cueing`) on/off	**CTRL**+D
	change `resLabelStyle` preference	**CTRL**+L
	change `resLabelStyle` preference	**CTRL**+A

MISCELLANEOUS	change background color	**CTRL**+E / CTRL+Q
	change "skin" color of the selected grob(s)	**CTRL**+E / CTRL+Q
	change "wire" color of the selected grobs	**ALT**+E / ALT+Q
	change display modes of the selected grobs	**CTRL**+X
	delete string label pointed by the cursor	**BACKSPACE**
	gui (graphical user interface)	**CTRL**+G
	drag the box	MiddleMB-click at boxCorner

Editing pairwise sequence-structure alignments

ICM has a powerful editor for pairwise and multiple alignments. ICM alignment editor robust and safe. It protects you from unintended changes in the alignment. To edit an alignment one only needs to select a block next to a gap and move it with arrows. In total, there are four operations one might need:

- select a block with one or several sequences to be moved (press Ctrl to add blocks). **Important:** since you can only move the selection **to the gapped space** , the moving front of the selection must be next to the gaps.
- (optional) create space on both sides around a vertical section of the alignment
- use the keyboard *arrows* to move the selected block with respect to the other sequences
- squeeze out the excessive gaps (an item in the alignment popup menu)

OPERATION	**KEYS**
set a vertical selection for ALL sequences in the alignment	**Double-Click**
add white space by hitting the Space bar	**SpaceBar**
remove white space	**Backspace**
select a sub-block for shifting	**Drag Left-Mouse-Button**
shift the selected block next to a gapped area	**Right and Left Arrows**

Constants

The values of most of the ICM-shell objects may also be represented explicitly in the ICM-shell as so called "constants" (i.e. in the `myFactors={1.2, -4., 5.88}` line, `myFactors` is an ICM-shell variable of the rarray type, while `{1.2, -4., 5.88}` is an "rarray" constant. The following constants are defined in the ICM-shell:

- **integers:** `-9999 12`
- **reals:** `12.0 -0.00003 2.`
- **logicals:** `yes no`
- **strings:** `"I see M", "Backslash (\\) and quote (\")" "line1\nline2"` or

```
## newlines are allowed between triple quotes, e.g.
a = """
```

```
 A text with lines
is also a string
"""
```

Escape sequences which can be used inside strings:

```
\a - bell
\b - backspace
\f - formfeed
\n - newline
\r - carriage return
\t - horizontal tab
\v - vertical tab
\\ - backslash
\" - double quote
```

- `integer arrays:` `{2, -1, 6, 0} {-8, -1, 2}` The comma is compulsory before a negative number, it can be skipped otherwise.
- `real array:` `{ -1.6 , 2.150 3., -160.}` Real arrays can also contain "ND" (Not Defined) and other special values. The following special values:
    - `ND` (not defined)
    - *>r_value*
    - *<r_value*
    - `INF` : infinity
    - `-INF` : infinity

  To compare an array with special value use the `Toreal` function:

```
read table csv "x_with_spec_values.csv" name="t"
t.A == Toreal({"ND"})
```

  Option `number` in `read table csv` will convert empty fields into `ND`However not all functions support them, be careful. Example:

```
 show Toreal({"ND"}
#
# here we are changing 0. values to NDs
 group table t {1. 0. 2. 0.}
 idx = Index( t.A == 0.0 )
 t.A[ idx ] = Toreal( Sarray( Nof(idx), "ND" ) )
```

- `string arrays:` `{"do","re","mi","fa","sol"} {"\n(newline), \t (tab)","\a (bell)"}`
- **`selections`** (find a detailed description below):

```
a_hiv?.  a_1,2.   a_*.            # objects
a_h*.a   a_m1     a_*.!w2,w15,z* # molecules
a_1.*/2:15,18:26 a_/18,his*     # residues
a_//ca,c,n a_1.c a_/2:4/!h*     # atoms
v_//phi,psi      V_2//?vt*      # variables
```

Subsets and Index Expressions

one can refer to an element or a subset of ten kinds of ICM-shell variables:

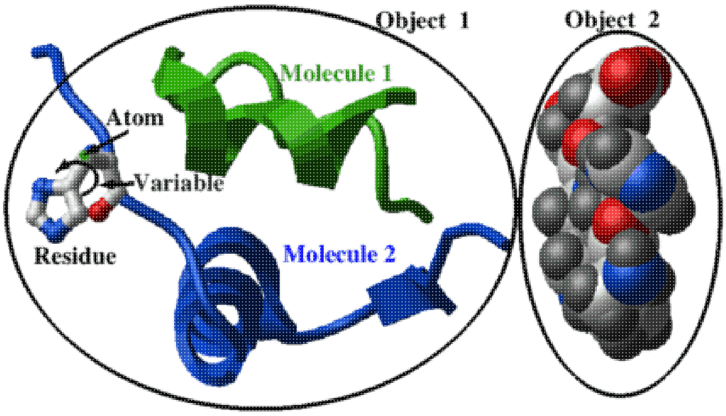
Variable type	**Expression**	**Result type**	**Example**
string	*string[i]*	*string*	`lastChar=str[Length(str)]`
	string[i1:i2]	*string*	`resName=pdbStr[18:21]`
iarray	*iarray[i]*	***integer***	`CurrSize=sizes[i]`
	iarray[i1:i2]	*iarray*	`frag=list[4:nitems]`
	iarray[I_]	*iarray*	`sublist=list[{1,3,5}]`
rarray	*rarray[i]*	***real***	`the=same[i]`

	rarray[i1:i2]	*rarray*	`all=as[for:iarr]`
	rarray[I_]	*rarray*	`Part=R1[{1,2,3}]`
sarray	*sarray[i]*	***string***	`best=menu[ibest]`
	sarray[i1:i2]	*sarray*	`fish=list[4:8]`
	sarray[I_]	*sarray*	`some=all[{1,2,3}]`
matrix	*matrix[i1*,i2]	***real***	`Element=M[4,5]`
	matrix[i1]	***rarray***	`atomCaVec=CoordMatr[15]`
	matrix[i1,i2:i3]	***rarray***	`thirdRow=M[3,1:5]`
	matrix[i1,?]	***rarray***	`thirdRow=M[3,?]`
	matrix[i1:i2,i3]	***rarray***	`firstColumn=mm[1:3,1]`
	matrix[?,i3]	***rarray***	`firstColumn=mm[?,1]`
	matrix[i1:i2,i3:i4]	***matrix***	`upperSqr=rot[1:2,1:2]`
sequence	*sequence[i]*	***string***	`amino4=bpti[4]`
	sequence[i1:i2]	*sequence*	`domain=seq[139:302]`
alignment	*alignment[i]*	*alignment*	`column4=globins[4]`
	alignment[i1,i2]	*string*	`AminoAcIn2ndSeq=globins[4,2]`
	alignment[i1:i2]	*alignment*	`motif=EFhand[15:27]`
profile	*profile[i]*	*profile*	`His=prof[18:18]`
	profile[i1:i2]	*profile*	`motif=prf[14:35]`
selection	*selection[i]*	*selection*	`ca18=ca[18]`
	selection[i1:i2]	*selection*	`frag=ca[14:35]`
table	*table[i]*	*table*	`show t[3]`
	table[i1:i2]	*table*	`delete t[3:5]`

Important note. When both lower and upper limits are explicitly specified, even if they are equal (e.g. *list* [3:3]), the type of the subset object remains the same. If only one element is specified, the rank may be reduced. The upper limit may be larger than the actual limit (e.g. t[3:9999]). You may also use `0` instead of the last element number (e.g. t[3:0]).

Molecule intro

Molecules are the main inhabitants of the ICM shell. The shell can contain many objects, each of which can be a soup (this expression belongs to my friend Gert Vriend) of separate molecules. Molecules, in turn contain residues and atoms. ICM can handle both **raw** objects, as they come from a PDB file or a mol-file, and a fully prepared for molecular modeling "ICM"-objects.



The non-ICM objects can be visualized, but they need to be `converted` into ICM-objects to perform the most interesting modeling operations. To specify the subsets of objects, molecules, residues, atoms and internal variables, you need to learn the language of molecular `selections`.
A quick preview of the selection language, using the picture above as an example:

```
display a_2. cpk           # object selection (the second object)
display a_1.1 ribbon green # molecule 1 from object 1
display a_1.2/his xstick   # residue his12 shown as balls and sticks
color   a_/1.2/12/n* xstick blue  # atoms: color nitrogens in blue
```

For an in-depth description of selections, read the next section.

Selections

Let us imagine that we decided to compare two structures deposited in the PDB. We will read both entries in the ICM shell, and define the following levels or organization. Each entry will form an **object**, each object will contain one or several **molecules**, protein molecules will naturally contain amino acid **residues** and residues will consist of **atoms.** Now, in the superimpose command, we will need to specify, or **select**, the molecules, residues or atoms which should be `superimposed`. The ICM shell language has a flexible way of selecting subsets of atoms, amino-acid residues, molecules, objects, as well as torsion angles and other internal geometrical parameters of molecules. Most of the ICM commands and functions dealing with molecules, for example, display, delete, minimize, etc., will operate on an arbitrary selection. What does a selection look like? For example, selection `a_2./2:14/c*` selects carbon atoms of residues from 2 to 14 of the second object. The general syntax of a selection is the following:
prefix **_** [*object(s)* **.**] *molecule(s)* **/** *residue(s)* **/** *atom(s) or variable(s)*
The object section including the *dot* (e.g. `1crn.`) may be omitted. In this case the selection will be performed in the `current` object.
There can be as many as five sections separated by **_ . /** and **/**,
Examples:

```
a_2ins.a,b/lys,arg/ca,cb,n*    # atom selection, '*' - any string
a_2ins.a,b/2:10/n,ca,c         # atom selection
v_crn./lys,arg/phi,PSI         # variable selection
```

(Note use of `PSI` torsion in the last example.)
Storing selections in named variables.
Selections can be assigned to a variable (e.g. `x = a_//c*`) and can be combined in an expression by *logical and* (**&**) or *logical or* (**|**), e.g. (`a_//n* & a_//ca`).

Selection Types

Three prefix types: a_ v_ and V_ . The Prefix defines one of the three selection types:

- atoms, residues, molecules and objects (`a_..`)
- free variables (`v_..`)
- all variables (`V_..`)

The a_ selection is the most popular and selects **atoms, residues, molecules** or **objects.** Therefore, there are four atom selection subtypes which are abbreviated as follows:

abbr.	**selection name**	**example**
*os_*	object selection	`a_ ; a_1. ; a_1crn. ; a_*.`
*ms_*	molecule selection	`a_1.2 ; a_a,b ; a_*.*`
*rs_*	residue selection	`a_/3:9 ; a_/* ; a_/"GKS"`
*as_*	atom selection	`a_1.2//ca,c,n ; a_//c*`

Two additional types of selections let you select amongst the free internal coordinates or all internal coordinates (both free and fixed). These selections are widely used in commands and functions related to energy minimization and sampling:

abbr.	selection name	example
vs_	selection from free internal variable	--{v_ ; v_1. ; v_1.2//x* ; v_2//?vt*}
Vs_	selection from all internal coordinates	--{V_ ; V_1. ; V_1crn.//!phi,psi,omg}

A selection can also be assigned to a named variable:
Example:

```
aa = a_//ca,c,n  # the backbone
show aa
```

The object and molecule sections are separated by a period, all other sections are separated by slashes. Inside each section, arguments in a list are separated by comma (,) while ranges are separated by colon (*from:to*).

Selection levels

There are four principal levels of selection: object selection, molecular selection, residue selection and atom or variable selection. The level is defined by the "lowest" section explicitly specified in a selection (e.g. `a_1.1/2:4` is a residue level selection, while `a_//ca` is an atom selection). These selections are referred to as ***os_ ms_ rs_ as_*** or ***vs_*** , respectively. If selection level is not important or the level is the lowest one (atoms or variables), selections are referred to as ***as_*** or ***vs_.***
The selection level of the interactive graphics selections is controlled by the `GRAPHICS.selectionLevel` preference. To change it from the command line, assign this variable to an appropriate level, e.g. `GRAPHICS.selectionLevel="atom"` .
Selection levels can be changed from the **GUI** interface, by changing the selection level

Examples

Examples of different selection levels (note that object and molecule names are arbitrary):

```
a_1,3.  a_mod*.   a_*.  a_"*benz?n*".   # object selections
a_3.mol1  a_zinc  a_$molNum  a_*.*      # molecule selections
a_/3:29,as?,ala  a_/*  a_*./"VHC?[!W]A" # residue selections
a//h?,c*  a_//T v_//phi,psi             # atom or variable selections
```

For example, `a_1,3.` is an object selection, and `a_/ala` is a residue selection.
Each section may contain a **negation** symbol **!** in the beginning. It selects *all, but the specified.* You can only use the negation symbol in the first position of a section and the negation will always apply to the *whole* section. For example, `a_/!ala,gly` is right, while `a_/ala,!gly` is wrong.
If object section together with the separating period is skipped, selection addresses the `current object` rather than all objects.

Select by number, range, name or pattern

Matching. Objects, molecules, residues, atoms and variables may be referred to by their names. Objects and molecules can be additionally referred to by their sequential numbers (e.g. `a_1.2)` **.** To select by a numerical name, use backslash before the name, e.g. `a_\123.` `Metacharacters`, such as *** ? []**, can also be used for pattern matching (e.g. `v_//?vt*)` **.**
Full syntax. A complete description of selection syntax for each level is as follows:

Object selection

(**a_** *obj.* or just **a_** for the `current object`):
a_ *name* **.**|(a_1crn. , note the dot at the end)
a_ *namePattern.*|(a_1c?n.)

a_ *relNumber*.|(a_2. means the second object)
a_ *num1:num2*.|(`a_2:5.` range from object 2 to object 5)

a_	the *current* object, it is a special case.
a_ " *commentPattern* ".	select by pattern matching in the object *comment* field.
a_ICM.	objects of ICM type (a_!ICM. - non-ICM objects)
a_NMR.	objects of NMR type
a_XRAY.	objects of XRAY type, see also `Select (a_*. , 'r`
a_CATRACE.	objects of `"Ca-trace"` type
a_SLIDE.	objects used in `slides`

Other object types (e.g. `"NMR"`,`"Fiber"`, etc.) can be selected or checked with the `Type` (*os_* 2) function.
Example:

```
read object s_icmhome+"all"
show a_   # the current object
show a_1,2:3.
show a_s1?.
show  a_"*Th[iy]o*".//!h*  #here we select by comment
set comment a_ "tag1 tag2 tag3, description"
show a_"*tag2*".
show a_"*tag2?tag3*".  # use ? for space
```

Molecule selection

a_*obj.mol*	in specified object(s),
a_*mol*	in the current object or
a_*.*mol*	in any object

by name:
a_*s_name* e.g. `a_m2` or `a_1.m2` in the current (`a_`), or the first (`a_1`) object, respectively. (Note that there is **no** dot at the end). If the name starts with a digit or one of the reserved one-letter types (see below), add backslash before the digit, e.g. `a_\123 , a_\A` .
by pattern
a_*s_namePattern* (`a_w*` - all water molecules in the current object)
by number(s)
a_*number* (a_2 , a_3.2,4,7) - relative number of molecule(s)
by range(s)
a_*num1:num2* (a_2:5 , a_2:5,10:12) - number range
by chemical formula (F):
a_**F***formula1*,**F***formula2*..
the chemical formula must be the same as the one returned by the ICM `String`(ms_) function **without hydrogens**, e.g.

```
  read pdb "1abe"
  show a_FC5O5  # selects 2 arabinose molecules
  String( a_2//!h* )
 C5O5
```

by special symbol for types of molecules:
a_*specialSymbol[,specialSymbol2..]*

- A peptides and proteins
- B molecules included in Biological unit
- C select by Chain, e.g. a_1.Cabc , use underscore ('_') for space.
- H hetatm, usually ligands and water molecules
- J*n1*[:*n2*] number of residues filter, e.g. `a_J3:5` 3 to 5 residues residues
- K molecules with linked sequence alignment
- L lipids
- M Metals
- N nucleic acids

- O*n1*[:*n2*] number of real atoms filter, e.g. `a_O3` for more than 3 atoms
- Q molecules which have a seQuence linked to them
- S sugars
- TRACE : molecules marked as "Amino" or "Nucl" with one to two numbers of atoms per residue
- W water including deuterated water (dod)
- U unknown (miscellanea)

Note that if a molecule name coincides with any of the above characters (i.e. `"ACHLMNQRSTUW"`), ICM gives preference to the type selection. To select by molecule name, use backslash (e.g. `a_1.\A` for chain named `"A"`)
Examples:

```
nice "1dnk"  # one peptide, two dna chains and other mols
a_A          # the peptide
a_N          # the two DNA chains
a_A,N        # the peptide and the DNA chains
rename a_1 "A"
a_\A         # chain NAMED "A"
read pdb "2ins"
delete a_W
read pdb "1e8s"
show a_TRACE  # shows Ca-trace molecules of two proteins and one RNA
```

Some special cases:

```
a_*   # all molecules in the current object
a_a   # molecule 'a' in the current object
a_.a  # molecules 'a' in all objects
a_*.a # the same as a_.a
```

selecting water molecules from pdb-files by their 'residue-field' number.
Water molecules in PDB files are numbered and the numbers are stored in the residue field. For consistency, we convert these numbers into residue numbers. At the same time the *names* of water molecules are built sequentially like this: `w1,w2,w3` . This way one can use both sequential numbering via molecule names and PDB-file numbering via residue numbers.

```
read pdb "1sri"
show a_w12:w15    # by molecule name, sequential numbering
show a_w*/719:721 # by original pdb number
```

converting any selection to molecules with the `Mol` function
Selection of any level, e.g. atoms, residues, and objects can be converted to molecules with the `Mol` (*selection*) function. Example:

```
Mol(Sphere(a_zinc a_1,2 8.)) # Sphere returns atoms
```

Residue selection

With respect to objects and molecules there are the following possibilities:

a_*obj.mol/res*	complete specification, (e.g. `a_*.*/14:19` or `a_2.3/ala)` .
a_*mol/res*	the current object and the specified molecules, (e.g. `a_w*/*`)
a_*/res*	all molecules of the current object, (e.g. `a_/23:25)`

Residue field specifications (for all molecules in the current object).
by name:
a_*/resName* (e.g. `a_/his` , or `a_/\001` - here we had to start with a backslash because the residue name looked like a number)
by residue name pattern:
a_*/resNamePattern* (e.g. `a_/as?` - asn or asp). A useful tip for DNA or RNA selections. Quite often bases are modified. To select A,T,G,C,U and their modifications, use `a_/??a` or `a_/??t` or `a_/??g` or `a_/??c` or `a_/??u`, respectively.
by residue number(s):
a_*/numChar* (`a_/3` or `a_/15A`) - PDB residue number may contain additional characters.
by residue range(s):

a_/*numChar1:numChar2* (`a_/4:15`,20:25) - reference residue number range
by amino acid sequence pattern:
a_/"*seqPattern*" (`a_/"G?GTE"`) - selects the fragment with matching amino acid sequence.
Example selecting all residues preceding prolines (the first expression selects dipeptides with the second proline, the second one excludes prolines):

```
 show a_/"?P" & a_/!pro*
```

by string and integer shell variables use the dollar substitution, e.g.

```
build string "ASDF"
i=2; j=3; a_/$i:$j
s = "12:13,15:19"
a_/$s/c*
```

Notice that value substitution for integer and string shell variables without the leading dollar symbol has been obsoleted.

by special symbols and expressions
by residue type
a_**/A** - residues of `"Amino" type` (N- and C-termini have different type) **displayed residues**

a_**/B** - barcode residues, see `Pattern(` *rs*). E.g. a_1.2/BL2LL . The gap lengths is calculated from the residue labels, see also the **Q** selection.

a_**/C** *resConservationCode* - selects residues by consensus letter, see `below`.
a_**/D** - displayed residues in the `ribbon` representation or with residue label

a_**/DR** - displayed residues in the `ribbon` representation only

a_**/DL** - displayed residues with residue labels
a_**/DD** - displayed residues in which either ribbon or some atoms are displayed
a_**/F..** selection by `site`, see `below`

residues identical to their homology target residues

a_**/Q** - barcode residues, see the **B** selection above. e.g. a_1.2/QL2LL . The gap lengths is calculated from the order of actual residues, the labels are ignored.

by secondary structure

a_**/S** *sec_struct_chars* - residues with certain `secondary structure` (e.g. `a_/SH` - only helices; `a_/SEH` - sheets and helices; `a_/S_` - only coil)
terminal residues (like N-terminal, C-terminal, and DNA 5' and 3' termini) a_**/T**

a_**/U** - unknown residues not described in ICM residue library
by alignment consensus
a_**/C** *resConservationCode* - selects residues according to the consensus of the *alignment* `link`ed to a molecule. The symbols can be combined, e.g. `a_/CYnh` for conserved tyrosines, negatively-charged residues and hydrophobics. Possible codes:

- A , C ... - particular conserved amino acid types (one-letter code)
- X - all absolutely conserved residues
- h - conserved hydrophobic residues (#)
- s - conserved small residues (^)
- p - conserved polar residues (~)
- o - conserved positive residues (+)
- n - conserved negative residues (-)
- a - conserved aromatic residues (%)
- x - not conserved but in the ungapped block (.)
- g - gap in one of the sequences of the alignment (' ')

(e.g. `a_/CXh` - selects all identities in the alignment and hydrophobic residues, `a_/CACg` - all conserved alanines, cysteins and gapped regions)
by functional features
a_**/F**[*SiteChars*] or a_**/F**"*siteID*" or a_**/F***local SITE.labelStyle*
residue selection by the one-letter site type or the site ID, respectively. Letter F refers to the word *feature* as

in the FT (feature table) field of Swissprot entries. The types along with their one-letter codes are listed in the `glossary` site entry. The default string, the `a_/F` selection, is defined by the `SITE.defSelect` string (you may redefine it), which defines important local features such as binding sites as opposed to domain-type sites such as signal peptides, zinc fingers and other protein domains. The PDB entries do not comply with the standard SWISSPROT site definitions, such as ACT_SITE BINDING etc., and are assigned by the user type **F** (selection `a_/FF`).
Example:

```
 nice "1as6"
 show site
 color ribbon a_/F magenta
 show a_/FF
 show a_/F"cu3"     # select only site named cu3
 show a_/F"MUTAGEN" # sites so defined in Swissprot
 set site a_1.1 "FT SITE 15 15 My favorite residue" label=2
 show a_/F2  # select by site label display style number
```

converting selections to residue level: The `Res` (*selection*) will convert any selection of higher level or lower level to the residue level. Example

```
 a_/SH & a_/pro  # a proline in a helix
 Res(Sphere(a_/pro 2.))  # expand to the neighboring residues
```

Atom selection

(**a_//***atoms*):
by name
a_//*name* (`a_.//ca` , `ca` is a usual name for alpha carbon)
by name pattern
a_//*namePattern* (`a_.//c*` for all carbons)
by special symbols and expressions
alternative atom positions in X-ray structures
a_//**A** *alterCharacter* - select alternative positions of the specified type (e.g. `read pdb "1cbn" ; show a_//Ab`). See also the `set comment "A"` *as_* command. This selection breaks down if an alternative has the character of one of the elements: Ac,Ag,Al,Am,Ar,As,At,Au . A newer (superior) form of this selection is `a_//:`*`char1char2..`* , e.g. `a_//c*:ab`

a_//**A** will select all atoms marked as alternatives (both main and secondary alternatives). This selection, in contrast to the explicit one (e.g. a_//:c) will also select the **unmarked** alternatives that are recognized as residues with the first coordinate less than 0.2A away form the same atom of the previous residue.

a_//**AS** will select only the **S**econdary alternatives (e.g. `color magenta a_//AS` . If you deleted a_//Aa atoms then a_//Ab become the main alternative and the other ones will become secondary. If you want to delete the primary, do not forget to clear the alternative flag with `set comment` *as* " " . The **AS** selection will also recognize the residues in the PDB file that are not marked by the alter character (see the a_//A description above). E.g.

```
 delete a_//AS  # delete secondary alternatives, do not need to clear
#
 delete a_//A & a_//!AS  # delete primary alternatives
 set comment a_//A " "   # clear the flag
 convert
```

by atom code: a_//C.. a_//CH a_//M
a_//**C***atomCodeNum[:atomCodeNum2]* - select by `atom code` as described in the icm.cod file, e.g. `a_//C2`,C4 selects aromatic and methylene hydrogens, `a_//C2:15` selects codes from 2 to 15 , e.g. `a_1.//C1:4,C101:115,C118A`
a_//**CH***atomHydrationCodeNum[:atomHydrationCodeNum2]* - by hydration/solvation code defined in `icm.cod` and `icm.hdt`

a_//**M***atomMmffCodeNum[:atomCodeMmffNum2]* - by mmff code e.g. `a_1.//M3,M10:15` . The atom types are described in `icm.cod` file.
displayed atoms, `a_//D..` * a_/**D**[*displayTypes*] - Displayed atoms (e.g. `a_//D` for all displayed atoms, or `a_//DWC` for wire or cpk). The following graphical types can be selected:

- **A** - labeled atoms
- **B** - `ball`
- **C** - `cpk`
- **D** - displayed atoms or atoms in displayed residues
- **T** - `tethered` atoms
- **S** - `skin` , **s** -all skin atoms including zero size.
- **V** - **V**an def Waals surface larger than zero, `solvent accessible surface` , **v** all surface atoms including zero area.
- **W** - `wire`
- **X** - `xstick` (i.e. ball or stick)
- *no arguments* - any graphical representation

Examples: `a_//DA , a_//DW , a_//DD`
Special named selections: as_graph graphically selected atoms:
`as_graph` selection contains graphically selected objects, molecules, residues, or atoms The level of selection depends on the `GRAPHICS.selectionLevel` preference. The level can be changed from the GUI interface or from command line.
strained atoms (atoms with high energy gradient)
a_//**G** - strained atoms (Gradient vector longer than `selectMinGrad`) You can also use the `display gradient` command.
Example:

```
build string "his trp trp"
display
randomize v_//phi,psi
selectMinGrad = 100.
show energy
display a_//G ball
display gradient
```

hydrophobic atoms a_//**H**
hydrogen bonding donors acceptors (one atom per residue at which the residue label is displayed)

a_//**HA** hbond acceptors including atoms of the following ICM types: (50:90,201,205:207,213,214,216,217,220:223,225,228:230,234:236,239:241,246,255,281:295)

a_//**HD** hbond donors.

a_//**E** donors and acceptors combined (includes non-aliphatic hydrogens and atoms of the following ICM types: (50:90,201,205:207,213,214,216,217,220:223,225,228:230,234:236,239:241,246,255,281:295)

a_//**I** donors and acceptors of the a_//E selection that are buried. This selection requires that the `show area` command is used beforehand.

residue label atoms (one atom per residue at which the residue label is displayed) a_//**L**
polar atoms a_//**P** defined simply as hydrogens connected to non-carbon atoms. We will tighten the definition in the future.

aromatic atoms a_//**R** It selects heavy atoms connected by aromatic bonds and hydrogens attached to them. Example:

```
build string "HWYP"
display skin
color skin a_//R magenta
```

tethered atoms
a_//**T** - `Tether`ed atoms (see also `a_//Z` - tether destination atoms)
tether-target atoms
a_//**Z** - `Tether` destination/target atoms (see also `a_//T` - tethered atoms) . A more general version of this selection is the `Select_lists`
chiral atoms a_//**X**[0123RLB] - chiral atoms. Each atom has two bits characterizing its chiral properties. If the two bits are presented as an integer, the chiral number has the following values:

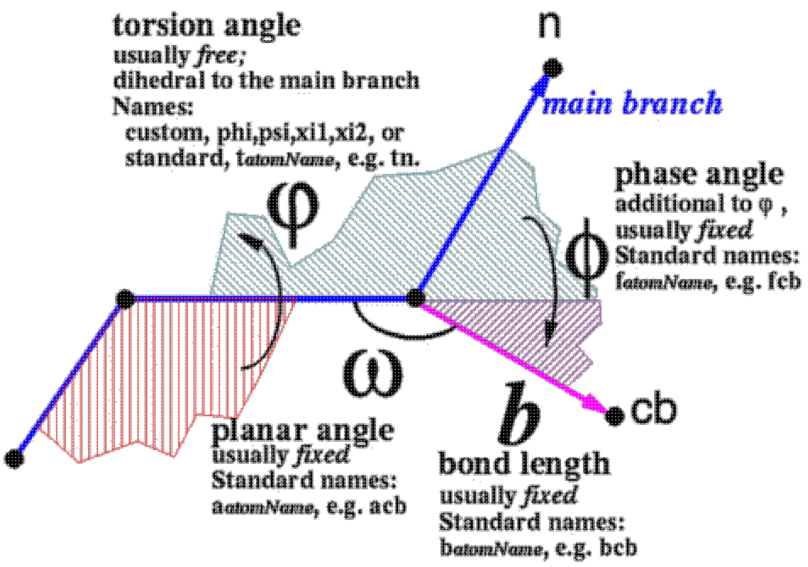
- zero - a non-chiral center
- 1 - a left topoisomer (L)
- 2 - a right topoisomer (R)
- 3 - a racemic mixture of both isomers (B)

The chiral symbols can be appended. For example `a_//X123` means `a_//X1 | a_//X2 | a_//X3` .
A short form of this selection, `a_//X` means all chiral atoms and is identical to `a_//X123` (or a_//!X0)
Examples: `a_m/3:4/X1` , `a_//XLR` (only left or only right chiral centers, but no racemic centers), `a_//XB` (only racemic centers)

See also: `V_//FC` to select chiral phase angles.
by absolute number
a_//*absNumber* - absolute number (all atoms of all objects are numbered sequentially starting from one)
converting to atom level: The `Atom` (*selection*) will convert any selection of higher level to the atom level.

Free and all variables (v_ and V_)

The ***v_selection*** selects **free variables** in molecular objects of ICM-type.
The main types of **internal coordinates** , or geometrical variables, are shown below:



The position of each atom branch is determined by the positions of the preceding atoms and three parameters: dihedral angle, planar angle and bond length. The dihedral angle for the main branch atom is the torsion angle itself, while for the secondary branch atoms the dihedral angle consists of the torsion angle plus the phase angle. The default fixation is given in the `ICM-residue library` and can be changed by `fix` and `unfix` commands. Individual free variables can be rotated interactively with `Ctrl-LeftMB-Atom-Click` and drag. A *vselection* can also be assigned to a named variable:
Example:

```
aa = v_//phi,psi  # the backbone torsions
unfix only aa
unfix only v_/10:15/phi,psi
```

V_ : selecting among all internal coordinates
Finally, the **V_** *selection* selects **both free and fixed variables** in molecular objects of ICM-type. You always need this type of selection in the `unfix` command. It makes no sense to unfix variables which are free already.
Here is a list of variable selection specifications:
by name:
v_//*name* (`v_//phi`)
by name pattern:
v_//*namePattern* (`v_//x*`) use asterisk ***** for any string, and question mark **?** for any character. Example: `v_//?vt*` selects the 6 "virtual" variables defining rigid body rotation and translation.
torsion variables
v_//**T***torsionCodeNum[:torsionCodeNum2]* - select by `torsion angle code` as described in the icm.tot file, e.g. v_//T11 selects the amide group torsion angle v_//T10:15 selects torsion codes from 10 to 15

angles (planar angle variables)
v_//**A***angleCodeNum[:angleCodeNum2]* - select by `planar angle code` as described in the icm.bbt file.

bond length variables
v_//**B***bondCodeNum[:bondCodeNum2]* - select by `bond length code` as described in the icm.bst file.
Displayed Variable Labels v_//**DL** - selects variables with displayed variable labels

Psi torsions not shifted to the next residue
v_//**PSI** - *psi* torsion angle which belongs to the residue you would expect. The reason for this definition is that from ICM point of the *psi* backbone torsion with rotation axis between Ca and C of residue *i* belongs to N-atom of the **next** residue *i+1* because N is the first atom this torsion angle moves. E.g., `v_/3/phi`,psi selection will contain the *psi* from residue 2 and then *phi* from residue 3. The definition **PSI** allows you to use the conventional attribution of angles, e.g. v_/3/phi,PSI is a pair of angles with axes around Ca atom or residue 3. **Important**. However, note that if you use selection expressions like `v_//phi,PSI & a_/2,3` it will not work (in contrast to `a_/2`,3/phi,PSI) and you will have to use the `Next` function.
Example:

```
 vPhi = v_/3/phi
 vPsi = v_/3/PSI
# BUT !!!
 vPhi = v_//phi* & a_/3
 vPsi = v_//PSI  & Next( a_/3 )
```

methyl group torsions
v_//**M** - torsion angles rotating Methyl-type terminal hydrogens (excluding polar hydrogen)
polar hydrogen torsions
v_//**P** - torsion angles rotating Polar hydrogens (e.g. hydroxyl group)
essential (non-hydrogen) torsions:
v_//**H** - side chain torsion angles rotating "Heavy" atoms
standard set of free torsions (excludes rings)
v_//**S** - all "Standard" free torsion angles as defined in the `icm.tot` file.
Note that `v_//M`, `v_//P`, and `v_//H` do not overlap, they are mutually exclusive. `v_//S` contains `v_//M`, `v_//P`, and `v_//H` as well as other standard torsion angles.
phase angles
v_//**F** - select **all** phase angles (usually they are fixed, so use `V_//F`)
V_//**FC** - select phase angles related to the **chiral** centers (see `set chiral` and `montecarlo chiral`)
all torsion angles
v_//**T** - select **all** free torsion angles, `V_//T` for all torsion angles including the fixed ones.

positional variables

v_//**V** - select **all** positional variables, 6-pack for each molecule or its part (see `convert` *rs_* E.g. `v_//V , v_2//V`

Functions returning selections

- `Acc` - select solvent-accessible atom/residues.
- `Atom` - convert to the atom selection
- `Deletion` - residues deleted according to the alignment
- `Insertion` - residues inserted according the alignment
- `Mol` - convert to the molecule selection
- `Name` - names of items, Name(.. full) returns selection strings of items
- `Next` - extract the next atom
- `Nof` - counts number of items in a selection
- `Obj` - convert to the object selection
- `Res` - convert to the residue selection
- `Sphere` - expand a selection by *r_radius* or 5ï¿½
- `Select` - selection of atoms according to their coordinates, bfactors, or other properties; healing selection gaps
- Res(Sphere(..)) will return residues in a sphere.

Substituting ICM-shell variables into a selection. You can insert the value of an `integer` or `string` ICM-shell variable anywhere inside your selection by using a **$** (dollar sign) prefix. (Note, this is a general ICM-shell substitution mechanism).
Examples:

```
 selstr="!w*/14:19"           # a string constant
 display a_$selstr
```

Logical operations. You can also assign selection to a variable, (i.e.: `backbone=a_//ca`,c,n) combine several selections using `logical operators` (example: `show a_/3:6 & backbone`).

Finding contiguous residue ranges with the String function

To identify contiguous ranges of residues in residue selection, use the `String` (*rs_*) function which will convert your selection into a string expression suitable for entering into a ICM-shell. For example, if we want to find all prolines surrounded by two other helical residues helical proline plus next and prev. residues we might do the following:

```
 read pdb "1dkf"
 rrange = String( a_/"?P?" ) # the result would look like "a_a.b/5:7,30:32"
 rg = Split(rrange,"/,|")    # split into sarray with {"a_a.b","5:7","30:32"}
                             # bar (|) helps with multiple chains
 okrg={""}
 k=0  # counter for good residue triplets with HHH and ?P?
 for i=2,Nof(rg)
   if Nof(Split(rg[i],":")) != 2 continue    # ignore molecular names
   if Sstructure( a_/$rg[i] ) == "HHH" then  # compare with ss-pattern
    k = k+1
     okrg[k] = rg[i]
   endif
 endfor
# now ok-ranges are stored in okrg string array e.g. {"5:7"}
# to use them Sum(okrg,",")
```

Regular expressions (regexp)

Functions supporting regular expressions:

- `Match` match expressions in a `string` or `sarray`
- `Replace` - replace expressions in a `string` or `sarray`
- `Index` - find substring position and length
- `Split` - by a regular expression

See `regexp syntax` .

ICM regular expression syntax

Simple expressions

- **.** any character except new line (to match anything, say **(.|\n)** or use **(?n)** in the beginning of the expression)
- **^** the beginning of the line
- **$** the end of the line
- **[*abc*]** any character from the list
- **[^*abc*]** any character NOT in the list
- **[*a-z*]** a range, e.g. [0-9] or [0-9A-Z]
- **\*c*** backslash suppresses special meaning of a character
- \\\\ backslash itself
- **(*string*)** enclose a simple expression in parentheses to write repetitions, back-references, or `field=`*number* expressions in the Split, Match and Replace functions.

Inline modifiers of regular expressions:

- **(?i)** ignore case until the end of the same enclosing group, e. g. 'aBc' ~ '(?i)abc', 'a((?i)bc)d' matches 'aBCd','abcd','aBcd', but not 'Abcd' or 'abcD'
- **(?-i)** match case-sensitive until the end of the same enclosing group, e. g. 'a(?i)bc(?-i)d' matches 'aBCd', but not 'Abcd' or 'abcD',
- **(?n)** begin matching newline character with dot '.': "1bc\nd2" ~ '(?n)1.*2'

Shortcuts

- **\d** matches a digit (`'[0-9]'`). `'\d+'` matches one or more digits.
- **\D** matches a NON-digit. `'\D+'` matches space between numbers
- **\w** matches a character in a word (`[a-zA-Z_]`). `'\w+'` matches a word
- **\W** matches a NON-word character. `'\W+'` matches the interword space
- **\s** matches a whitespace character, or a separator (`[ \r\t\n\f]`)
- **\S** matches a non-separator symbol
- **\b** matches a word boundary, i. e. a boundary between \w and \W symbols, for example, `'\bedgeh\b'` matches inside `'the edge'` and does not match inside `'the hedge'`

Repetitions and back-references

(*a* and *b* are simple regular expressions, e.g. a DNA base `[ACTG]`, or `([hp]anky.*)`):

- *a*? - nothing or a single occurrence of *a*
- *a** - nothing or any number of repetitions of *a*
- *a*+ - matches *a* at least once or more
- *a*{**n,m**} - matches *a* from n to m times
- *a*|*b* - matches *a* or *b*
- *ab* - matches *a* **and** *b*
- (*a*)**\1** - \1 is a back-reference: matches *a*, then matches exactly the same string. Back-references can go from \1 to \9.

A problem with the posix repetitions

Imagine that you want to match text between two tags, e.g. `<i>one</i>` in a text which has two items of the same kind (`<i>one</i> and <i>two</i>`). Unfortunately, we can not just use `<i>.*</i>` to match `<i>one</i>` since the POSIX standard tries to match the MAXIMAL LENGTH expression between the italic tags (shown in bold are the flanking expressions: **`<i>`**`one</i> and <i>two`**`</i>`**). A straight-forward solution of this problem is to make a more complex definition of the word between the tags, by saying that the 'italized' word should not contain the '<' symbol.

ICM followed Perl in using the question mark (**?**) after the repetition symbol to enforce the *minimal* match. The minimal match expressions will look like this (*a* is a simple regular expression, like a character or a string in parentheses):

- *a*?? - nothing or a single occurrence of minimal occurrence of *a*
- *a**? - nothing or any number of repetitions of minimal occurrence of *a* (e.g. `Match(s,'tag(.*?)endtag':n))`
- *a*+? - matches *a* at least once or more

Therefore:

- `'<i>.*</i>'` - matches the entire '`one</i> and <i>two`'
- `'<i>[^<]*</i>'` - explicitly prohibits the tag inside. matches only the first word
- `'<i>.*?</i>'` - the '`*?`' expression enforces the smallest match

Parsing XML example: DrugBank.

The DrugBank database is a unique bioinformatics and cheminformatics resource that combines detailed drug (i.e. chemical, pharmacological and pharmaceutical) data with comprehensive drug target (i.e. sequence, structure, and pathway) information. The database contains 6826 drug entries including 1431 FDA-approved small molecule drugs, 133 FDA-approved biotech (protein/peptide) drugs, 83 nutraceuticals and 5211 experimental drugs. Additionally, 4435 non-redundant protein (i.e. drug target/enzyme/transporter/carrier) sequences are linked to these drug entries. Each DrugCard entry contains more than 150 data fields with half of the information being devoted to drug/chemical data and the other half devoted to drug target or protein data. Read more information: `here`

The most complete drug information (target, transporter, carrier, and enzyme information) is provided in XML format. Chemical structures are provided separately in SDF format

The following example will demonstrate how to deal with such data in ICM.

1. **Read the XML data directly from the website**

   ```
   read xml "http://www.drugbank.ca/system/downloads/current/drugbank.xml.zip" name="drugb
   ```

   The command above will create `collection` object "drugbank".
2. **Examine the content**

   ```
   icm/def> Name( drugbank )
   #>S string_array
   drugs
   ```

   This shows us that collection contains a single root node called "drugs"
3. **Going further gives the following:**

   ```
   icm/def> Name( drugbank["drugs"] )
   #>S string_array
   drug
   partners
   xmlns
   xmlns:xs
   xs:schemaLocation
   icm/def> Type(  drugbank["drugs","drug"] )
   array
   icm/def> Type(  drugbank["drugs","partners"] )
   collection
   icm/def> Name(  drugbank["drugs","partners"] )
   #>S string_array
   partner
   icm/def> Type(  drugbank["drugs","partners","partner"] )
   array
   ```

   Which means that drugbank["drugs","drug"] is an `array` where each entry contains the information about particular drug. In addition there is an another `array` drugbank["drugs","partners","partner"] which contains an additional information about targets.
4. **Examine individual entries**

   ```
   drugbank["drugs","drug"][1]
   drugbank["drugs","drug"][2]
   drugbank["drugs","partners","partner"][1]
   drugbank["drugs","partners","partner"][2]
   ```

   The default output format for displaying `collection` is `JSON` which gives you nicely formated easy-to-read text. Looking at the output it's easy find the fields of interest.

   WARNING: do not try to show the entire array into the terminal window because it'll take very long and most likely you'll need to kill the window.
5. **Fetching individual fields**

   Let's create a table with a single column containing an array with drug cards.

   ```
   add column drugs drugbank["drugs","drug"]
   ```

   Hint: In GUI you can resize all simultaneously by holding 'CTRL' key which resizing an individual row.

   The single field can be extracted by providing dot separated path to it. Note that fields which contain non-alphanumeric characters must be quoted.

   - A.drugbank - OK
   - A.'drugbank-id' - must be quoted

   ```
   # extracts drugbank-id into separate column
   add column drugs function="A.'drugbank-id'" name="drugbank_id"
   # extracts name into separate column
   add column drugs function="A.name" name="name"
   ```
6. **Fetching multi-value fields**

   Multiple properties will be extracted as an array for each drug entry.

   ```
   # display targets information for the second entry
   drugs.A[2]["targets","target"]
   # extract array of partner IDs for each drug into separate column
   ```

```
add column drugs function= "A.targets.target.partner" name="partner_id"
Type( drugs.partner_id[2] )  # array
```

This way to extract multiple properties has one problem. For entries with only one property the result will be not array but rather individual value (E.g: Type(Type(drugs.partner_id[1]). This will prevent from the unified access to the column in the future. In such cases it's recommended to use ':' operation instead of '.'. The result of this operation will always be an array (even for single entries).

```
delete drugs.partner_id
add column drugs function="A.targets.target:partner" name="partner_id"  # will create a
Type( drugs.partner_id[1] )  # array (even for single entries)
```

7. **Querying XML fields**

   Let's say you want to extract a value of the property with name which start with "logP". It can be done similar to the `ICM-table filtering operations`. The only difference is that colon ':' (instead of dot) must be used to separate field name

   The general filtering syntax:

   ```
   <field1>.<field2>:<queryField> <op> <value>
   ```

   The following operations are supported in array filtering: **==,!=,>,=,<=,~,!~**

   Example:

   ```
   # query and extract logP property
   add column drugs function="(A.'experimental-properties'.property:kind ~ '^logP').value[
   ```

   Note that some entries contain text information ('0.61 [HANSCH,C ET AL. (1995)]') so the result column will not be automatically converted to `rarray`. You can convert it explicitly:

   ```
   # empty or 'bad' entries will be marked as 'ND'
   add column drugs Rarray( drugs.logP ) name="logPNum"
   delete drugs.logP
   ```

   The other example will extract Wikipedia links:

   ```
   add column drugs \
     function="(A.'external-links'.'external-link':resource == 'Wikipedia')[1].url"\
     name    ="wiki"
   ```

8. **Joining with information from drugbank["drugs","partners","partner"]**

   For each drug entry we have list of partner IDs which refers to information from drugbank["drugs","partners","partner"] `array`. To `join` them we need to add this array to the other table and extract fields which will be used in join.

   ```
   # creates a table and put partner entries there.
   add column partners drugbank["drugs","partners","partner"]
   # extract ID column which will be used to join with drugs.partner_id
   add column partners function= "A.id" name="id"
   # extract uniprot-id from the "external-identifiers" array using query functions
   add column partners \
    function= '(A."external-identifiers"."external-identifier":resource ~ "UniProtKB")."id
    name    = "uniprot_id"
   ```

   Finally we need to join **drugs.partner_id** with **partners.id**.

   ```
   join drugs.partner_id partners.id column ="drugs.*,partners.uniprot_id" name="drugs"
   ```

   Note that since drugs.partner_id contains multiple entries for each row the result drugs.uniprot_id will also contain multiple entries for each row. You can set special format with `set format` command to execute a special action when particular uniprot entry is clicked.

   ```
   # load sequence
   set format drugs.uniprot_id \
    "<!--icmscript name=\"1\"\nread sequence swiss \"http://www.uniprot.org/uniprot/%1.txt
   # or simply go to the website
   set format drugs.uniprot_id "<a href=http://www.expasy.org/uniprot/%1>%1</a>"
   ```

9. **Joining with chemical structures** The final step would be to add a chemical structure information.

```
# read SDF from the website
read table mol "http://www.drugbank.ca/system/downloads/current/structures/all.sdf.zip"
# join 'mol' column
join drugs.drugbank_id drugs_chem.DRUGBANK_ID  column="drugs.*,drugs_chem.mol" name="dr
```

A little bit more rearrangements and your table is ready to be exported to SDF file.

```
move drugs.mol 1  # move structure column to the first position
delete drugs.A    # delete drug-card information
delete drugs.partner_id # delete partner id information
write table mol drugs "mydrugs.sdf"
```

See also: `collection`, `read xml`

Hierarchical cluster trees

The records, or rows, of any `table` can be clustered into a hierarchical tree, and one or several trees associated with this table can be stored with it, displayed and edited in the ICM GUI, and deleted.

A tree is created with the `make tree` command. We can decide 1) the tree type and, 2) the distance function between two table rows, as well as establish a number of arguments. Then a tree object is added to the header of the table and is stored together with the table. The table gets a new column with the tree order, and optionally two new elements: and a column with the branch number at a certain level, (option `split)` and the distance matrix (option `matrix)` .

The related commands and functions:

`make tree` create tree object and attach it to the table
`Split` function to split cluster by threshold or number of clusters
`split` command to change the position of tree cursor (separator) and recalculate new cluster numbers
`Name( table.cluster i_tree [index,label,matrix,sort,split] )` names of important table columns
`Max`(table.cluster) the distance of the root node
`Distance` of the cluster splitting level
`Nof( table.cluster tree )` clusters
`Centers` of clusters
Example:

```
# create a distance matrix
m=Matrix(5,3)
m[2,1:3]={1. 0. 0.}
m[3,1:3]={1. 1. 0.}
m[4,1:3]={1. 1. 1.}
m[5,1:3]={1. 0.1 0.1}
D = Distance( m )

# create a table and move distance matrix into header
group table t { "a" "b" "c" "d" "e" } "label" {1. 2. 2. 1. 4. } "val"
group table t append header D "dm"
make tree t distance = "dm"       # uses external distance matrix for clustering

# get cluster number with threshold set to the middle
cl = Split( t.cluster, Max( t.cluster )/2 )
add column t cl name="cl"

# group by cluster and take rows by smallest value of "val" column
group t.cl t.val "min" all "refmin" name="t1"
```

Selecting N representatives from clusters

This involves several steps:

- creating a tree and a table column with cluster numbers
- selecting cluster representatives according to a certain threshold in the cluster tree

Example:

```
read table mol s_icmhome + "drug_groups.sdf"
make tree drug_groups
I = Index( drug_groups.cluster center 0.4 ) # divide at threshold 0.4
```

Arithmetics

Most of the ICM-objects can be used in arithmetical, logical of comparison expressions. In this section we describe operations defined in the ICM-shell.
Members of the arithmetic, logical and comparison expressions
Abbreviations: `integer` (**i**), `real` (**r**), `string` (**s**), `logical` (**l**), `iarray` (**I**), `rarray` (**R**), `sarray` (**S**), `matrix` (**M**), `sequence` (**seq**), `profile` (**prf**). `alignment` (**ali**), `map` (**m**), `graphics object (grob)` (**g**), `atom selections` (**as**), `selections of internal coordinates`, for example torsion angles, (**vs**) , and `table` (**T**). Table arrays are abbreviated as **T.I, T.R** and **T.S**, depending on the type

Assignment

allows you to assign a value to a variable.
Syntax: ***ICM-shell-variable-name = Value or expression***
If the name is new, a new ICM-shell variable is created , an object of the matching type will be overwritten.

Examples:

```
 a=1            # create new integer variable  a
 b=a*a          # create variable b as product a*a
 c=a*Sin(45.)   # create new real variable c
```

Chain assignment for the logical system variables and semi-colon separated commands. Exressions like `l_info=l_commands=l_warn=no` are allowed for the logical system variables

Also, from version 3.6-1 several semi-colon separated commands can be specified in one line.

Assignment fo selected elements of an array:

array[I_indices] = *Value*

array[I_indices] = *Matching_array_of_values*

Several elements in integer, real and string arrays can be assigned at once to a single value , or, element-by-element to a matching array. Example:

```
a = {1 2 3}
a[{1 3}] = { 3 1 }
a[{1 3}] = 10
```

Unique name. To find a unique name for a new variable, use the `Name(` *"nameRoot"* , `unique`) function.

Assignment and operations **in place** are also possible and it allows to modify an existing variable rather than create a new one. Example:

```
i += 1  # adds one to i, better than i = i + 1
s += " and more .. "
I //= 15  # append to an iarray
R //= 1.5 # append to an rarray
S //= "one more element" # append to an sarray
```

Arithmetic operations

The following operations are defined in the ICM-shell:

- **addition** (+) :
  - **i+i** returns **i** (e.g. `2+3` returns `5`) ,
  - **i+r, r+i, r+r** return **r** (e.g. `2+3.` returns `5`) ,
  - **I+I** returns **I** (e.g. `1,2`+{2,3} returns `3,5`),
  - **I+R, R+I, R+R** return **R** (element by element),
  - **s+s, s+i, s+r** return **s** (i.e. "what" + "If" returns "whatIf","file"+2 return "file2"),

- ♦ **S+S, S+I, S+R** return **S** (the above three operations for each element),
- ♦ **M+M** returns **M** of the same dimensions (element by element addition),
- ♦ **prf+prf** returns **prf**,
- ♦ **grob + R3** return **grob** with coordinates translated by **R3**,
- ♦ **map+map, map+i, map+r, i+map, r+map** returns **map** of the same dimensions.

• **subtraction** (**-**) :
- ♦ **i-i** returns **i**,
- ♦ **i-r, r-i, r-r** return **r**,
- ♦ **I-I** returns **I**,
- ♦ **I-R, R-I, R-R** returns **R** (element by element),
- ♦ **M-M** returns **M** of the same dimensions (element by element subtraction),
- ♦ **map-map, map-i, map-r, i-map, r-map** returns **map** of the same dimensions.

• **multiplication** (*) :
- ♦ **i*i** returns **i**,
- ♦ **i*r, r*i, r*r** returns **r**,
- ♦ **I*I** returns **I** (element by element, e.g. `{1,2}*{3,4}` returns `3,8`),
- ♦ **I*R, R*I, R*R** return **R** (element by element). The **scalar product** is returned by `Sum(R_1`,R_2), and the **vector product** is returned by `Vector(R_1`,R_2) (two 3D vectors)
- ♦ **M*M** returns a matrix product of the two matrices (M[nk]*M[km]==>M[nm]),
- ♦ **M*R, R*M** returns **R**,
- ♦ **prf*prf** returns **prf**,
- ♦ **map*r, map*i, i*map, r*map, map*map** return **map** (operations on each element),
- ♦ **grob*r, grob*i, r*grob, i*grob** return **grob** with transformed coordinates.

• **division** (**/**) :
- ♦ **i/i** returns **i** (integer division, e.g. 3/4 returns 0),
- ♦ **i/r, r/i, r/r** return **r** (real division, e.g. 3/4. returns 0.75),
- ♦ **I/I** returns **I** (integer division of elements),
- ♦ **I/R, R/I, R/R** return **R** (real division of elements),
- ♦ **map/r, map/i** return **map** (operations on each element),
- ♦ **grob/r, grob/i** return **grob** with transformed coordinates.

* **concatenation, appending into array** (**//**) :
- ♦ **i//i** returns **I[2]** (e.g. `2//3` returns `2,3`),
- ♦ **r//r** returns **R[2]** (e.g. `2.2//3.3` returns `2.2,3.3`),
- ♦ **s//s** returns **S[2]** (e.g. `"a"//"b"` returns `"a","b"`),
- ♦ **I//i** returns **I[n+1]** extended by the integer (e.g. `1,2`//3 returns `1,2,3`),
- ♦ **I//I** returns **I[n+m]** (e.g. `1,2`//{3,4} returns `1,2,3,4`),
- ♦ **R//r** returns **R[n+1]** extended by the real (e.g. `1.,2.`//3. returns `1.,2.,3.`),
- ♦ **R//R** returns **R[n+m]** (e.g. `1.,2.`//{3.,4.} returns `1.,2.,3.,4.`),
- ♦ **S//s** returns **S[n+1]** extended by the string (e.g. `"a","b"`//"c" returns `"a","b","c"`),
- ♦ **S//S** returns **Sn+m** (e.g. `"a","b"`//{"c","d"} returns `"a","b","c","d"`),
- ♦ **M**[n,m]//**M**[n,k] returns **M[n**,m+k] (matrix concatenation row by row),
- ♦ **seq//seq** returns concatenated **seq** (similar to s+s);
- ♦ **prf//prf** returns concatenated **prf;**
- ♦ **grob//grob** returns concatenated **grob** (similar to s+s);
- ♦ **parray//parray** returns concatenated **parray;** (this applies to different types of pointer arrays, e.g. **chem_array//chem_array**)

* *`ali1//ali2`* returns **projected alignment.** Projected concatenation of two alignments sharing the same sequence. The shared sequence serves as a ruler for merging the two alignments. The alignments can be of arbitrary size and number of sequences. In the simplest case of three sequences `a`, `b`, `c` and alignments `ab` and `bc`, the operation `ab//bc` will create an alignment of three sequences `a b c` . The function `Align(ab//bc`,{1,3}) will extract the, so called, projected alignment of **a** and **c** through **b.** Example:

```
  ali1    //    ali2  returns   Projected ali.
a VYRWA-W     b FK-WG--KW     a  VYR-WA---W
b -FKWGKW     c AKGWAPGKW     b  -FK-WG--KW
                              c  -AKGWAPGKW
```

Additionally, character arrays (strings) can be projected from sequence to alignment and back with the `String` (..) function and numerical residue properties can be projected from sequence via alignment with the `Rarray` (..) function.

Logical operations

Logical operations with `table` arrays are described separately (see table in Glossary).

- **and** (**&**):
  - ♦ **l & l** returns **logical**, e.g. `yes & no` returns `no`
  - ♦ **as & as** returns selection **as** with objects molecules residues present in both initial selections, (e.g. `a_2//ca & a_//T` returns the tethered Ca atoms of the 2nd molecule),
  - ♦ **as & s**, multiplication by a **string mask**, e.g. a_//ca & "x-" returns the *odd* Ca atoms.
  - ♦ **as & seq** returns **residue sub-selection** of **as** with the **matching sequence**, e.g. a_*. & 1crn_m returns residues matching the crambin sequence.
  - ♦ **as & R** returns **atom sub-selection** with coordinates within a six dimensional box array **R** (see also function `Box`) , e.g.

    ```
    read pdb "1crn"
    display ribbon
    color ribbon green Res( a_//* & {0.,0.,0.,9.,9.,9.} )
    ```

    More types of selections are returned by the `Select`, `Sphere`, and `Acc` functions.
  - ♦ **vs & vs** returns selection **vs** of internal coordinates present in both initial selections, (do not forget that v_ are free variables, and V_ are all variables);
  - ♦ **vs & as** returns subset of initial variables **vs** which is related to selection **as**, e.g. side-chain torsion angles in the sphere around loop 14:18 can be selected as follows:

    ```
    V_//xi* & Sphere( a_/14:18 )
    ```

  `multiplication` comments on logical multiplication of two selections below.
- **or** (**|**) :
  - ♦ **l|l** returns **l** (e.g. `yes|no` returns `yes`),
  - ♦ **as|as** returns **as** with members of both selections (e.g. `a_/4:6 | a_//ca`)
  - ♦ **vs|vs** returns **vs** with variables from both selections (e.g. `v_//phi,psi | v_/3`)
  - ♦ **vs|as** returns **vs** is equivalent to `vs | (V_//* & as)`
- **not** (**!**) :
  - ♦ **!l** returns **logical** negation to the argument (e.g. `!yes` returns `no`),
  - ♦ **!as** returns **aselection** of the same level with members not included in the selection argument (e.g. `!a_//ca`)
  - ♦ **!vs** returns **vselection** with variables not included in the selection argument (e.g. ! v_//phi,psi)

  Negation can also be applied to each section between slashes of *as_* or *vs_*. E.g. `a_//!h*` (all non-hydrogens).

In place operations.

ICM-shell variables can be modified in place by using the following operators:

operator	**function**	**data types**	**example**
+=	add in place	i,r,s	i += 1
-=	subtract in place	i,r	r -= 2.2
***=**	multiply in place	i,r	r *= 2.
/=	divide in place	i,r	r /= 2.
//=	append an element to an array	I,R,S,P	`t.Name //= "Jack"`

If a variable does not exist yet, this operation will create the variable and assign a type according to the right-hand operand.

Comparison operators

Most of them are true comparison operators and return `logical yes` or `no.` In comparisons of table arrays or string arrays with strings, the comparison returns a subtable or subarray, respectively. Comparison operations with the `table` arrays are described separately (see table in Glossary).

- **equal** (**==):**
  - ♦ **i==i, i==r, r==i, r==r, s==s, I==I, R==R, S==S, M==M, as==as, vs==vs**, exact equality of two objects;
  - ♦ **p==i, p==s** return **l.** Test the value of an ICM-shell `preference` . Example:

    ```
    if(wireStyle==1) print "int. or string is ok" # or
    ```

```
if(wireStyle=="chemistry") print "double bonds"
```

- ♦ **S==s**returns **S**, a sub-sarray of elements exactly matching **s** (e.g. `"aa","b","aa","c"=="aa"` returns `"aa","aa"`, see also **S ~ s**),

• **not equal** (**!=**) :

- ♦ **i != i, i != r, r != i, r != r, s != s, I != I, R != R, S != S, M != M, as != as, vs != vs** inequality of two objects;
- ♦ **p!=i, p!=s** return **l.** Test an ICM-shell `preference` . Example:

```
if(wireStyle != 2) print "No chemistry, sorry"
```

- ♦ **S**!=**s** returns **S**, a sub-sarray of elements not matching **s** (e.g. `"aa","b","aa","c"!="aa"` returns `"b","c"`, see also **S !~ s**).

• **greater than** (**>**) :

- ♦ **i > i, i > r, r > i, r > r**
- ♦ **s > s** lexicographic comparison for sorting ("apple" &lt "orange")

• **less than** (**<**) :

- ♦ **i < i, i < r, r < i, r < r, s < s**

• **greater or equal** (**>=**) :

- ♦ **i >= i, i >= r, r >= i, r >= r, s >= s**

• **less or equal** (<=) :

- ♦ **i <= i, i <= r, r <= i, r <= r, s <= s**

* **fuzzy-equality, inclusion or `pattern matching`** (**~**):

- ♦ **s**~**s**returns **logical** `yes` if string matches a pattern.
- ♦ **S**~**s**returns **S** of sarray elements matching the pattern **s.** This comparison is similar to the UNIX **grep** command; it returns a subarray of lines matching the pattern rather than `yes` or `no`. Do not forget to add flanking asterisks (*) if the pattern occurs in the middle of a string. Example:

```
show {"abc","bcd","ee"} ~ "*[be]?"
# Another example
read database s_icmhome + "foldbank.db"
        # sarray SE contains sequences
CxCseqs = SE ~ "*C?C*" # all strings containing C?C pattern
```

• **fuzzy-not-equal** (**!~**) :

- ♦ **s !~ s** returns **logical** `yes` if string does not match a pattern **s.**
- ♦ **S !~ s** returns **S** of sarray elements **not** matching the pattern **s.** S!~s is similar to the UNIX **grep -v** command; it returns a subarray of lines not matching the pattern.

Advanced operations and some comments

1. Integers are automatically converted to reals in binary operations containing both integers and reals. However, in expressions like *integer1* **/** *integer2* (the same for iarrays) they are *not* converted into reals and the result will be different from what you might expect. For example, `3/4` returns `0`, but `3/4.` returns `0.75`.
2. In **s+i, s+r, S+I, S+R** expression numbers are automatically converted into strings. In the s+s expression the second string is simply appended to the first one. Examples:

```
show "one " + "two"  # result: "one two"
file = "aa"+ 4       # result: "aa4"
show {"a","bb"} + {1.2,3.2}  # result: {"a1.2","bb3.2"}
```

3. Selection arithmetics. The level of the expression *as_1 & as_2 & as_3 ...* or *as_1* | *as_2* | ... (the same with *vs_*) is defined by the **lowest level** selection in the chain (atoms - the lowest < residues < molecules < objects). For example, in an expression **a_/10 | a_2/15/cg** the second selection is an atom-level-selection and the first one is a residue-level one. The result is the atom selection of all atoms of residue 10 plus *Cg* atom from residue 15.
4. **Selection logically multiplied by string, array, or mask** Multiplication of a selection to a string-mask or sequence. The resultant selection inherits level of the first argument. The mask is applied periodically to switch off some of the selected elements. For example mask "0001111" will switch off the first three elements in every seven. The 'switch off' characters may be the following: ' ' (space),'-','0'. Example masks to switch off the third element of five: `"xx xx"`, `"11011"`, `"++-++"` . Operations upon the sequence will select only the fragment with the specified sequence from the original selection. Multiplication by an array of 6 numbers {x,y,z,X,Y,Z} selects atoms within the specified box. Example:

```
read object "crn"   # load crambin object
rs_ = a_/11:15      # define residue selection rs_
rs_ = rs_ & "xx xx" # switch off the third element (res. 13)
display cpk a_//* & {1. 0. 1. 5. 7. 6.} " # a box
```

5. **Transitional (or projected) alignment** Projected concatenation of two alignments sharing the same sequence. If two-sequence `alignments` share the same sequence, they may be merged with the shared sequence as a ruler. In the simplest case of three sequences a, b, c and alignments ab and bc, the operation ab//bc will create an alignment of three sequences a b c. The function `Align(ab//bc`,{1,3}) will extract the, so called, projected alignment of **a** and **c** through **b.**

Examples of expressions:

```
i = i1/i2 + (i3-r4)*2.5/Pi

l_results=(l_beer & l_wine & !l_snacks) | l_vodka
if (l_results & n_glasses >= 4) print "Hangover.."

for i=1,215        # list streets of Manhattan north from Houston
  print "Street " + i
endfor

prices = { 25. 6. 12.6 }
tips   = { 4.  1. 2.   }
print prices + tips        # the result is { 29. 7. 14.6 }
```

Flow control

ICM contains a complete set of control statements to allow looping, jumping and conditional branching.

Loops

Two types of loops are allowed, namely *for-loop* and *while-loop.*
For-loop

```
for  <i_index> = <i_from> ,  <i_to> [, <i_increment> ]
 ...
 ...
endfor
```

While-loop

```
while( <logical_expression> )
 ...
 ...
endwhile
```

Examples:

```
for i = 1, 9
  print "ICM-shell proudly announces that i=" i
endfor

for i = 1, 4
  print "ICM-shell proudly announces that i=" i
  for j = 1, 3
    print "ICM-shell proudly announces that nesting is possible and j=" j
  endfor
endfor

read object "crn"
for i = 1, Nof(a_/*)  # Nof(a_/*) means 'the number of residues'
  print Label(a_/$i)
endfor

i = -2
while (i != 4)
  i = i+1
  print i
endwhile

while(yes)
  print "endless loop, please wait 8-)"
```

```
 endwhile
```

Any number of nested loops may be used.

Conditional branching

Several types of conditional statements are allowed in the ICM-shell.
if

```
 if ( <logical_expression> ) <command>
```

if-then-endif

```
 if ( <logical_expression> ) then
    ...
    ...
 endif
```

if-then-elseif-..else-endif

```
 if( <logical_expression> ) then
     ...
 else
     ...
 endif
```

or

```
 if ( <logical_expression> ) then
   ...
 elseif ( <logical_expression> ) then
   ...
 elseif ( <logical_expression> ) then
   ...
 else
   ...
 endif
```

Note: `end if` or `else if` (instead of `endif` or `elseif`) are not accepted by ICM-shell.
Examples:

```
 JohnnySaid = "The gloves didn't fit"
 if ( JohnnySaid == "The gloves didn't fit" ) print "You must acquit"
#
 grade = "bad"
 if (grade == "excellent") then
   print "It's great!"
 elseif (grade == "good") then
   print "It's good!"
 elseif (grade == "bad") then
   print "It's not so bad!" # do not be harsh on your kids
 endif
```

Jumps

Three types of jump controls are possible, namely commands **break**, **continue** and **goto**. **break** interrupts the loop, **continue** skips commands until the nearest **endfor** or **endwhile** and continues looping, and **goto** jumps to any point below.
break

```
 <for-loop> or <while-loop>
     ...
     if ( <logical expression> ) break
     ...
 <end of loop>
```

continue

```
 <for-loop> or <while-loop>
     ...
     if ( <logical expression> ) continue
     ...
 <end of loop>
```

goto

```
 ...
 if ( <logical expression> ) goto <label>
 ...
 ...
 <label>:
 ...
```

Examples:

```
    for i = 1, 6
      print "currently i=", i, "and it will be increased at the next step"
      if (i == 3) then
        print "... but at this point we should stop it, sorry..."
        break
      endif
   endfor
   print "end of the loop demonstrating *break*, bye"

    for i = 1, 6
      if (i == 3) then
        print "... let us skip over step 3 and continue looping"
        continue
      endif
      print "currently i=", i, "and it will be increased at the next step"
   endfor
   print "end of the loop demonstrating *continue*, bye"

   for i = 1, 5
     if (i == 3) then
       print "... but at this point we decided to skip 3-rd step, sorry..."
       goto A
     endif
     print "currently i=", i, "and it will be increased at the next step"
A:   print " "
   endfor
   print "end of the loop demonstrating 'goto', bye"
```

Note: *go to* (instead of *goto)* is not accepted by the ICM-shell. Any combination of alphanumeric characters beginning with a letter (upper or lower case) may serve as a **label.** Also keep in mind that *goto* can jump only **forward;** the **backward** *goto* is not allowed.

ICM molecular objects

An ICM molecular object represents one or several molecules which can coexist in physical space, so that the energy of the molecular system can be calculated. For example, if you have two homologous molecules superimposed, multiple conformations of the same structure such as NMR structure determinations or alternative positions of a side chain, they must belong to different objects. The number of objects that may be loaded in ICM is limited only by the available computer memory. Objects may be of several types (see also: the `Type` (*os_* `2`) function):

- `"ICM"` - the only complete type which is good for everything including energy calculations
- `"X-Ray"` - incomplete (stripped) object created by `read pdb`. The structure is determined by X-ray crystallography. Good for graphics and geometrical analysis
- `"NMR"` - incomplete (stripped) object, structure determined from NMR data, similar to the "X-ray" type above.
- `"Model"` - incomplete (stripped) object, theoretical model also similar to the "X-ray" type above.
- `"Ca-trace"` - incomplete (stripped) object, only alpha-carbon atoms.
- `"Simplified"` - simplified representation.

ICM-molecular objects are created from residues and molecules described in the ICM `residue library`. Its content (sequences and names of molecules) is specified in an `ICM sequence file` (see also `IcmSequence` function). An ICM-object can also be created from a non-ICM object (e.g. of X-Ray type) with the `convert` command.

Energy and Penalty Terms

The energy function calculated for any conformation of an ICM molecular object consists of individual terms described in this section. For most of them ICM calculates analytical derivatives which use gradient minimization. The terms can be switched on and off with the `set terms [only] "xx,yy,.."` command, e.g.

```
 set terms "el"          # activate electrostatic term
 set terms only "vw,14" # reactivate only "vw" and "14" terms
```

Existing terms are returned in `s_out` after the `show term` command, or returned by the `Info` (term) function.

The following commands also understand shortcuts for groups of energy terms:

- `show energy`
- `minimize`
- `montecarlo`

The list of shortcuts:

- "energy","ecepp","ecep","ey" is equivalent to "vw,14,to,hb,el,ss"
- "map" or "mp" returns a set of terms according to the existing maps. If a map with a suitable system name is found, the terms is activated (see also `Info` (map)). The following map names trigger the corresponding term activation:
  "m_gh","m_gc","m_gb","m_ge","m_gs","m_g1",..,"m_g5"
- "mmff" is equivalent to "bs,bb,af,vw,14,to,hb,el,ss"

van der Waals ("vw")
nonbonded interatomic pairwise interactions (1-5 and further, i.e. two atoms separated by more than 3 covalent bonds). If not for tests, this terms should always be used with the "14" energy term which considers 1-4 interactions. The `ECEPP/3` force field is used. Parameters are specified in the `icm.vwt` file and are taken from `Momany et al., 1975`. Both the usual 6-12 term and a soft van der Waals terms are available. See also: `vwMethod`, `vwSoftMaxEnergy`, `vwCutoff` .
1-4 van der Waals ("14")
A part of the total van der Waals energy for atoms separated by exactly three covalent bonds. Repulsion for 1-4 pairs is cut in half according to the ECEPP energy function. This term is complementary to the `"vw"` term and is usually used with the `"vw"` energy term.
Hydrogen bonding energy ("hb")
A different form of the `"vw"` term (10-12 instead of 6-12 for `"vw"`) for hydrogen bonding donors and acceptors as specified in `icm.cod` and `icm.hbt` files. Parameters are taken from `Momany et al., 1975`. The electrostatic contribution to a given hydrogen bond is not included in "hb" and is calculated as part of the electrostatic energy.
The cutoff distance for hydrogen bonding interactions is controlled by the `hbCutoff` parameter.
Torsion energy ("to")
dihedral angle deformation energy K*(1+-cos(n*Phi)). The parameters K, sign and n are given in `icm.tot` file. Parameters are taken from `Momany et al., 1975`,
Electrostatic energy ("el") This term is calculated in four different ways depending on the value of `electroMethod` preference. If `electroMethod="boundary element"` the solvation component is in `r_out` and the envelope surface area in `r_2out` .
A special case: if the van der Waals energy is calculated with the `vwMethod ="soft"` , the electrostatic energy will be automatically buffered to avoid singularities. You will see that the electrostatic term `"el"` changes upon switching from `vwMethod=1` to `vwMethod=2` . The buffering artifically increases the distance between two charged atoms to avoid having negative energy values better than the van der Waals repulsion and, therefore, will prevents collapse of oppositely charged atoms.

1. A simple electrostatic energy (`electroMethod="Coulomb")` . The Coulomb law is used to evaluate the energy. The `dielectric constant` is constant.
2. the distance dependent electrostatics (`electroMethod="distance dependent" ;` *currentDielConst* = `dielConst` * *DISTANCEij*) Advantage: this term has analytical derivatives and can be used in local energy minization.
3. A better electrostatic free energy (`electroMethod="MIMEL")`, uses the `Modified IMage ELectrostatics` approximation (`Abagyan and Totrov, 1994`) to evaluate both the

internal Coulombic energy and electrostatic polarization free energy. Disadvantage: this term has no analytical derivatives and has no effect on local energy minimization. It can be a part of the energy function in global optimization such as `montecarlo` or `ssearch` . The solvation component is stored separately in `r_out` . `REBEL` provides a more accurate evaluation of the electrostatic solvation energy. For small molecules, use `mimelDepth = 0.3` (default `0.5`).

4. The most accurate electrostatic free energy: (`electroMethod="boundary element"`) which uses so called `boundary element method` to solve the Poisson equation to calculated a electrostatic free energy of a protein surrounded by a continuous aqueous solution. In addition to the total energy, one can extract the two components: the electrostatic solvation energy from `r_out` , and the Coulomb energy can be calculated as a difference between the total electrostatic energy and r_out.

Surface term ("sf"). Map `m_ga`
Surface energy is based on atomic solvent-accessible `surfaces`. Depending on the `surfaceMethod` preference this term is either a surface tension which is evaluated as a product of the total solvent accessible area by the `surfaceTension` parameter (currently 0.012 kcal/mole/A^2) or is a product of atomic accessibilities by the atomic energy density parameters similar to those proposed by `Wesson and Eisenberg (1992)` (check `icm.hdt` file). The "sf" term is evaluated at each Monte Carlo or systematic search step, but not during local minimization (we do not calculate analytical energy derivatives).
The atomic accessible surfaces are calculated using a faster modification of the `Shrake and Rupley, (1973)` algorithm where the `surfaceAccuracy` parameter defines the resolution. This algorithm analyzes all atom neighbors for each atom and Sometimes a part of molecular system is represented with the grid energy terms (`"gc"`, `"gh"`) rather than by explicit atoms. In this case the atomic accessibilities need to be corrected.
This correction can be introduced with a special map, called `m_ga` which stores implicit neighbor information from the parts represented with the grid potentials. The `m_ga` map is calculated with the `make map potential "sf" ..` command (see the `make map potential` command), along with other grid maps.
The surface term can be weighted with the `sfWeight` parameter and is affected by the `surfaceAccuracy` parameter (set it to 5 for higher accuracy).
Entropic free energy term (conformational entropy of side-chains) ("en")
Configurational entropy of side-chains is evaluated on the basis of their maximal possible entropy which is read from the `residue library`. Note that this term is calculated at room temperature (300 K), so that the ICM-shell variable `temperature` does not affect the entropic contribution (see `Abagyan and Totrov, 1994` for values) and solvent-accessible area of a side-chain.
Phase angle bending term ("af")
Harmonic term U*(f1-f0)^2. Parameters U and f0 are taken from `icm.bbt` file. Sometimes referred to as *improper torsion.*
Bond stretching energy ("bs")
Harmonic term U*(b1-b0)^2. Parameters U and b0 taken from `icm.bst` file.
Distance restraints ("cn") a penalty term restraining two atoms to a certain distance range. The shape of the potential is *soft square well* with lower and upper bounds. This term may be used to determine three-dimensional structure from a set of interproton distances (NOEs) resulting from NMR experiments. There are local and global distance restraints (drestraints). Local restraints become weaker and vanish as the distance grows (similar to the van der Waals forces), while global restraints become stronger as you deviate further from the required distance range.
See also files: `icm.cnt` and `icm.cn` .
Disulfide bonds and covalent bridges ("ss")
a penalty term establishing the additional (extra-tree) covalent bridges. Currently there are three types of covalent bridges: disulfide bonds, peptide bonds and thioester bonds. In each case several distance constraints are imposed to enforce the correct covalent geometry. The constraints for the disulfide bonds include Sg1-Sg2, Sg1-Cb2, Sg2-Cb1, Cb1-Cb2 atom pairs. The extra CO-NH bond involves C-N, C-H, O-N and O-H constraints. Similarly, CO-SH bond involves C-S, C-H, O-C, O-H, C-C and O-H constraints. The functional form of this penalty term is identical to `local distance restraints`. The disulfide SS bonds are automatically formed when you load the object. The disulfide bonds may be LOCAL, i.e. when two sulfur atoms feel each other ONLY at small distances. See also: `icm.cnt`, `disulfide bond`, `make disulfide bond`, `make peptide bond`, `delete disulfide bond`, `delete peptide bond`.
Tethers ("tz")
Quadratic restraint E= `tzWeight` *Distance^2 between atoms in the current object and static atoms in a different object (as opposed to distance restraints "cn" between atoms in the same object). The target value of the distance is zero. See also: `read pdb`, `set tether`, `term ts` , and `tether` .

Tethers to Self ("ts")
Term `"ts"` is used in minimization to temporarily tether the atoms specified in the `selftether=` *as_* argument of the `minimize` or `montecarlo` command to their initial coordinates. The advantage of this

term that you do not need to have any other objects. To self-tether a fraction of atoms, use the selftether= *as_* option of the minimize command.

Example:

```
  build string "lys"
  randomize v_//x*
  minimize "vw,to,ts" selftether=a_//ca,c,n
```

See also: TOOLS.tsWeight , TOOLS.tsToleranceRadius , term tz , set selftether, delete selftether , selftether

Multidimensional variable restraints ("rs")

Energy associated with multidimensional ellipsoidal attraction zones (in which dimension they look like soft square wells with flat bottom) in a hyperspace of internal variables (e.g. preferred side-chain or backbone torsion angles). Vrestraints are defined in icm.rst and icm.rs files and are earmarked to be used in energy calculations (as opposed as for the BPMC) with the rse field (as opposed to rs). Use set vrestraint energy command to assign vrestraints. Described in Abagyan, Totrov and Kuznetsov, 1994 (pp. 494,495).

Density correlation ("dc")

Penalty function associated with correlation between the static map (the current map is used by default) and a virtual map generated from atomic positions on the fly. The dcMethod preference allows you to choose between several different functional forms of this term:

$DC = 1 - Sum(D_i - < D >)(A_i - < A >)/(N * Rmsd(D)*Rmsd(A))$

and $DC = 1 - Sum(D_i - < D >)(A_i - < A >)/ N$

where D_i is the map value, and A_i is the density generated dynamically from atomic positions.

The term has analytical derivatives with respect to the internal coordinates and can be efficiently locally minimized. By adding this term one can combine energy minimization with the real space fitting into electron density.

A more detailed description can be found in the dcMethod section.

Crystallographic correlation between Fobs and Fcalc ("xr")

van der Waals grid potential for carbon probe ("gc")

van der Waals interaction between explicit non-hydrogen atoms of an ICM object and a van der Waals potential calculated on the grid. To calculate this term one needs an ICM object and map named m_gc which is calculated with make map potential "gc" ... The calculation also counts the number of atoms in the area with Evw > 0.8 * GRID.maxVw and stores this number in r_2out .

By default the make map potential "gc" command will create two maps: m_gc map for a carbon probe, and m_gl map for atoms with the van der Vaals radius larger than 1.8 (e.g. sulphur or phosphorus). With the "gc" term on both maps will be used.

Note that these two maps, m_gc and m_gl are very similar, but one is calculated **for** a carbon like probe, while the other for a sulphur-like probe and, therefore, is an inflated version of the m_gc map.

van der Waals grid potential for hydrogen probe ("gh")

hydrophobic potential ("gs")

electrostatic grid potential ("ge")

Calculates the electrostatic potential contribution from the atoms specified in the make map potential as_ command. The contributions are calculated by the Coulomb formula with distance dependent-dielectric constant ($4*D_{ij}$)

hydrogen bonding grid potential ("gb")

property grid potential ("gp").an atom **property** term that can carry up to 7 different grid maps. The grid maps are generated with the make map potential "gp" command and are controlled by the GRID.gpGaussianRadius parameter. The atom type projection is defined by the set type property command. The relative weight of each map of the gp term (g1,g2,...) is controlled by the gpWeights parameters. Term "gp" represents seven maps:

- g1 : hydrogen bond donor field
- g2 : hydrogen bond acceptor field
- g3 : sp2 hybridization field
- g4 : lipophilicity field
- g5 : large-size atom field
- g6 : positive and negative charge
- g7 : electronegativity/electropositivity field

Potential of mean force ("mf" and pmf)

Note that term name is "mf", while icm keyword for some commands is pmf

The mean-force "mf" potential was designed as a generic energy term which is calculated for pairs of atoms according to their pmf-types and inter-atomic distances. The definitions of the pmf-types and energy-distance dependencies for each contributing pair of atom types can be loaded from a .pmf pmf-file. To read this file use the following command.

```
read pmf "icm.pmf" # or any other mf-file
```

The list of pmf-interacting pairs is calculated dynamically and only the pairs at smaller that vwCutoff threshold distance are considered. **Note:** It is important that vwCutoff = 9.5 is used in binding score evaluation.
There is a preference called mfMethod which controls if the atoms in the same molecule can interract. By default only intermolecular pairs of atoms are considered (mfMethod = 1). Switching mfMethod to 2 (or "all") allows one to include all atomic pairs regardless of which molecule they belong to in the "mf" term calculation.
Since this term is quite general one can prepare different pmf-parameter files for solving different problems. The default file icm.pmf has been derived from receptor-ligand complexes and allows pmf-scoring of docked ligands. Another file: ident.pmf was designed to specify attraction of the same atom types and allows one to solve a problem of chemical superposition.
The relative weight of the pmf-term is controlled by the mfWeight parameter.
An example in which we evaluate a binding score:

```
read object "rec"
read object "anwers1"
move a_2. a_1.
vwCutoff = 9.5
mfMethod = 1
show energy "mf" a_1 a_2
e = Energy("mf")
```

An example in which flexible superposition of two molecules is performed:

```
 build string "his ; gly trp"  # two molecules
 read pmf "ident.pmf"
 fix v_//omg
 display
 superimpose a_1 a_2
 vwCutoff = 2. # mf uses vwCutoff to calculate lists
 montecarlo "mf" v_2//?vt*  | v_//!?vt*  # internal variables + positional for the second mole
```

See also: mfMethod , pmf-file, mfWeight .

Integer shell parameters

Here is the alphabetically sorted dump of integer parameters defined in the ICM-shell. These parameters are used by various commands and functions and can be changed interactively, e.g.

```
 mncallsMC= 10000
 montecarlo
```

ICM-shell integer variables are the following.

autoSavePeriod

In the course of a montecarlo or ssearch procedures which may run for days, the current stack of conformations which accumulates the best energy representatives of different conformational areas is saved periodically to allow access to intermediate results of the simulations. The above parameter defines the number of stack changes after which it is saved to a disk file. Set autoSavePeriod to 1 if you want to be conservative.
If you set autoSavePeriod to 0 , the stack will **not** be saved at all.
Default (10).

defSymGroup

defines a crystal space group number. To find the group name and symmetry operations use the `Symgroup` function. Default (0) means that the group is not defined.
Examples:

```
defSymGroup = 19  # direct assignment. You know group 19, don't you?

defSymGroup = Symgroup("P212121") # Oh, you do not! ..

defSymGroup = Symgroup("P61 2 2") # This one you do not remember for sure
```

i_out

an integer where some commands or functions store their integer output:

- `Rmsd` saves the number of aligned equivalent points;
- `Srmsd` saves the number of aligned equivalent points;
- `convert` saves the number of heavy atoms missing from the pdb-template (e.g. atoms of the flexible lys side-chain are not given in the pdb-file).
- `superimpose` saves number of aligned equivalent points;
- `set tether` saves the number of tethers imposed;
- `set drestraint` saves the number of distance restraints imposed;
- `set vrestraint` saves the number of variable restraints imposed;
- `make disulfide bond` saves the number of imposed disulfide bonds;
- `minimize` saves the number of function evaluations;
- `montecarlo` saves the total number of function evaluations during minimization;
- `show area skin` saves the total number of triangles in the Connolly construction.

Default (0).

i_2out

the second variable for additional integer output. (see also `i_out` and `I_out`)

maxColorPotential

local electrostatic potential in kcal/e.u.charge units at which the surface element is colored by extreme red or extreme blue. All higher values will have the same color. This absolute scaling is convenient to develop a feeling of electrostatic properties of molecular surfaces.
If the `maxColorPotential` is set to `0.` the `color grob potential` command will perform automated scaling to the absolute maximal value of the potential.

See also: `color grob potential`, `dsRebel` , `Potential` , `make grob potential` .
Example:

```
build string "se glu arg"  # dipeptide
maxColorPotential = 3.
dsRebel a_ yes
maxColorPotential = 6.
dsRebel a_ yes
```

maxMemory

maximal memory size requested by the program in megabytes. It is used to read blocks of databases in the search commands. Make sure that this parameter is reasonable. If your maxMemory is larger than what your computer actually has, expect serious delays. However, usually computers can handle it by swapping memory onto disk, which can be slow.
Recommendation: divide your available RAM by a factor from 2 to 4. Current memory resources are reported by the chkdsk command on a PC or by the top command on a UNIX workstation. Do not forget that ICM itself will *additionally* allocate some `BufferSpace` specified in the `icm.cfg` file.

Default (10.0) Mb

---

minTetherWindow

maximal number of preceding torsions strictly speaking rigid bodies which are locally minimized during the chain growth procedure (the `minimize tether` command) to create an ICM-object with ideal geometry on the basis of a set of arbitrary atom coordinates (often referred to as the `regularization` procedure).
Default (30).

---

mnSolutions

this parameter limits the number of hits retained by the program after a search. It is used in several icm-search functions:

- `find molecule` - chemical substructure search
- `find pattern` - find sequence pattern in sequences of mol. objects.
- `find database` - advanced sequence similarity search
- `align` *ms_1 ms_2* - alternatives solutions for 3D superposition
- find profile - find protein Prosite profiles in a sequence
- `find prosite` - find protein Prosite patterns in a sequence

Default (100).

---

mncalls

maximal number of function calls in local minimization performed in `minimize`, and as a part of one step of a multistep procedure in `montecarlo`, `ssearch`, `convert` . The number of function evaluations required to find the local minimum varies widely depending on the terms used (i.e. the `"tz"` term makes minimization very slow, if structure is far from its target). If the minimum is found according to the `tolGrad` criterion, the procedure will be terminated anyway.
Default (100).
See also: `minimizeMethod` , `tolGrad` , `drop` .

---

mncallsMC

maximal number of function calls in the `montecarlo` command. Since the procedure performs random steps accompanied by local minimization (controlled by the `mncalls` parameter), the number of function evaluations for the whole procedure can be roughly evaluated as a product of `mncalls` and the number of MC iterations. `mncallsMC` should be sufficiently large to ensure convergence of the global optimization procedure and may range from 10,000 for a single side-chain, 100,000 for a 3-4 residue peptide to several million calls for 15-20 residue peptide or a large protein loop.
Default (`1000`). The default value is small to minimize damage of the unintentional calls of the montecarlo command.
See also: `montecarlo` , `mncalls` .

---

mnconf

maximal number of `conf`ormations in the conformational `stack` . The stack stops growing after this number is achieved and starts replacing representative conformations with higher energy values by new conformations with superior energies, if the latter are found.
Default (50)
See also: `montecarlo` , `ssearch` .

---

mnhighEnergy

maximal number of consecutive accepted trial conformations which do not change the conformational `stack` because their energies are higher than energies of the stack conformations. Therefore, the `montecarlo` procedure is walking in the high energy area and is probably wasting its time. When this threshold is reached the procedure acts according to the `highEnergyAction` parameter.
Default (50)
See also: `mnvisits` , `mnreject` , `stack` .

mnreject

maximal number of consecutive rejections (due to the Metropolis criterion) of trial conformations generated by the `montecarlo` procedure. When this threshold is reached the procedure acts according to the `rejectAction` parameter (which usually increases the simulation temperature).
Default (10)
See also: `mnvisits` , `mnhighEnergy` .

mnvisits

maximal number of visits to the same slot of the conformational `stack` in the course of a `montecarlo` procedure. When this threshold is reached the MC procedure acts according to the `visitsAction` parameter. A visit is an event when a newly generated conformation finds a slot with a similar conformation in it, but the stack conformation is not replaced by the new one because it has a better energy. The optimal *mnvisits* parameter grows with the size of the problem (it may be several hundred for a 15-20 residue peptide).
Default (50)
See also: `mnreject` , `mnhighEnergy` .

nLocalDeformVar

Number of backbone torsion angle variables (excluding omegas) which are changed simultaneously to provide local deformation. This parameter can be less than the actual number of backbone torsion angles in the loop. In other words it is OK if the loop contains more than *nLocalDeformVar* variables, however, if it contains less than *nLocalDeformVar* variables, it will not be deformed.
Default (10), minimal number (8).
See also: `montecarlo local`.

nSsearchStep

number of steps per variable for `ssearch` . Normally the whole [-180., 180.] range is divided into `nSsearchStep` parts. In the `local` mode (i.e. the search is performed around a particular conformation) the total search range around each variable is defined by the `ssearchStep` parameter (30. deg. by default)
Default (3) .

nProc

This variable can be used as a hint to run an ICM command in parallel if possible. The current list of ICM commands which supports this option is below:

- `set charge formal auto`
- `enumerate tautomer`
- `learn type="nn"`
- `Score`
- `Distance chemical`

Example:

```
read mol table "big.sdf" name="t"
set charge formal auto nProc=8    # run in 8 threads
```

randomSeed

is a seed used by the random-number generator in the `montecarlo` , `randomize` , `Random` function. Helpful if you need to *reproduce exactly* a calculation which uses random number(s). If the variable has its zero default value, the random function is seeded from the current time plus the process id. Otherwise, if you explicitly redefine it before, let us say, a `montecarlo` run, it will use the specified number.
Note that the `randomSeed` parameter can be set only once **in the very beginning of the session**. If you redefine its value in the middle of the session, it will not be used. To push the new value of the seed, use the `set randomize i_newRandomSeed` command.
Default (0).
Examples:

```
 randomSeed=2493059372  # this number you took from the previous run
 montecarlo             # simulation will reproduce the previous one
#
...
#
 set randomize 2493059372
 montecarlo
```

segMinLength

secondary structure `segments` shorter than this threshold will be ignored when a simplified quantitative representation of the polypeptide fold is constructed using the `assign sstructure segment` command.
Default (3).

sequenceBlock

length of the contiguous sequence block in sequence output.
Default (10).
See also: `sequenceLine` .

sequenceLine

maximum sequence length printed on each line. Usually sequence is additionally subdivided into smaller blocks.
The same parameter also controls the size of alignment block as saved by the `write alignment` command.
Example:

```
 read alignment s_icmhome+"sh3"
 sequenceLine=1000
 write sh3 "aaa"
```

Default (60). Values >= 1 .
See also: `sequenceBlock`

surfaceAccuracy

integer accuracy level used in surface calculations (not graphics) and boundary element electrostatics. By reducing the level, you can speed up the accessibility calculation in the `show area surface` command. It may be important to increase `surfaceAccuracy` to 5 in the `rebel` and `make boundary` calculations. The corresponding number of dots per sphere is the following:

- Level 1 (89 dots)
- Level 2 (144 dots)
- Level 3 (233 dots)
- Level 4 (377 dots)

- Level 5 (610 dots)

Default (3)
See also: `show area surface`, `"sf" energy term`.

---

windowSize

number of elements used for sliding window averaging by the `Smooth` function.
Default (7).

Real shell variables

ICM-shell real variables are the following.

---

addBfactor

additional B-factor which may be added to the current atomic B-values to create a smoother electron density map from a set of atoms. See also:

- `make map factor`
- `make map cell`

Default (0.0)

---

alignMinCoverage

a threshold for the ratio of the aligned residues to the shorter sequence length. All alignments shorter than *alignMinCoverage*minLength* will not be reported by `find database` command.
The default value is 0.5. However the parameter can be tuned with the respect to the database and the nature of the query sequence.

- Search against the protein domain sequence database: use 0.5 or higher
- Search a multidomain sequence against long multidomain sequences: use 0.1 or lower

See also: 'alignMinMethod , `find database` .
Default (0.5)

---

alignOldStatWeight

a parameter influencing the statistical evaluation of sequence comparison significance in the `find database` command.
Statistical significance can be evaluated in two ways: first, *a priori*, i.e. before the database search and based only on the individual score of an alignment of interest and its *theoretical* distribution, or, second, *a posteriori*, i.e. on the fly and on the basis of all *empirically* observed scores of *other* alignments in the course of the database search.
The parameter ranges from 0. to 1. and sets how two different statistical criteria of alignment significance, a precomputed (the old one) and a run-time, should be mixed. Zero corresponds to only the run-time measure (the *new* way) in which the significance is evaluated on the run-time statistics of the observed alignment scores, while one corresponds to the statistics evaluated before the search using the formula from `Abagyan and Batalov, 1997` . If the database is small then the run-time score statistics may be incomplete and `alignOldStatWeight` closer to 1. is a better choice. On the other hand, the run-time statistics has several principal advantages:

Precomputed statistics (1.) based on individual alignment score and length	**Run-time statistics (0.) based on distribution of scores**
works always	relies on database diversity

is trained only in 64 condition sets and ZEGA alignment	automatically adjusts to any set of conditions, e.g. `gapFunction`, or `alignMinCoverage`
does not reflect compositional bias	automatically reflects all seq. properties
does not reflect extra terms to the score	accounts for solvent accessibility correction (see `accFunction`)

The run-time statistics will fit the scores to an optimized empirical function. This function avoids the problems of the normal distribution, and certain pitfalls of a popular EVD function. The resulting P-value is a reliable estimate of the false positive rate if the database is sufficiently diverse, i.e. the fraction of sequences similar to the query is small. For example, searching a tyrosine kinase through a database of tyrosine kinases will yield incorrectly low pP-values (pP = -Log(P)).
Reliable expect-values: P * Nof(sequences) <= 0.1 .
Example:

- Swissprot has N=89,000 sequences. LogN = 4.95
- Reliable pP = LogN + 3, twilight pP is from LogN + 1. to LogN + 3.

axisLength

length (in Angstroms) of the X,Y,Z axes of the coordinate frame. The axes can be displayed by the `display origin` or `display virtual` command. The axes are marked **X Y Z** . Example:

```
build string "ala ala his his"
display
axisLength=10.
display origin
```

Default (1.5)

clashThreshold

a clash is defined as an interatomic distance less than a sum of van der Waals radii of two atoms of interest multiplied by the `clashThreshold` parameter. For hydrogen bonded atoms, the distance threshold is additionally reduced by `20%` .
See also: `display clash` , `show clash`, `GRAPHICS.clashWidth`
Default (0.82)

cnWeight

weighting factor for the interatomic `distance restraints` penalty term. See also: `tzMethod` , `drestraint` and `Bfactor` .
Default (1.0)

consensusStrength

a real parameter between 0. and 1. controlling the percentage of sequence identity in an alignment column required to establish a consensus. This parameter `r` is applied to the C0 parameters defined in the `CONSENSUS` table according to this formula:

```
C = C0 + (100-C0)*2*(r-0.5)
```

The `0.5` value corresponds to the percentage in the table. Default (0.5)

dcWeight

weighting factor for the density correlation term "dc".
Default (1.0)

COLOR.bg : background color in 3D graphics

user preference for the background color (overwrites `icm.clr` preference) E.g.

```
COLOR.bg = "grey"
write system preference
```

See also:

- `COLOR.distanceAtom`
- `COLOR.labelAtom`
- `COLOR.labelResidue`
- `COLOR.labelSite`
- `COLOR.labelVar`

COLOR.distanceAtom : default colors of interatomic distances

interatomic `distance` is shown by a dotted line of this default color. These colors and styles can be changed individually.

COLOR.label... default colors of labels.

- COLOR.labelAtom : atom labels
- COLOR.labelResidue : residue labels
- COLOR.labelSite : site labels
- COLOR.labelVar : variable labels

they overwrite the colors specified in `icm.clr`

CONSENSUS_strength

regulates the strength of consensus modifying the CONSENSUS.fraction values. The CONSENSUS table controls the rules of consensus derivation from an `alignment` . This table may look like this:

```
#>T CONSENSUS
#>-symbol------fraction----residues---
   A           80          A
   C           90          C
   D           85          D
   d           60          ND
...
...
```

The CONSENSUS_strength (denoted ***S***) parameter can increase or decrease the `fraction` values ***f*** according to the following formula: ***f1 = f + (100-f)*(S-0.5)*** Therefore if ***S*** = 1. all fraction values become equal to 100% . This affects the `Consensus` function and the GUI representation of alignment consensus in ICM versions above 3.0.
To color structures according to the consensus, use the `color alignment` *rs*_command, or, interactively, left-click on a color icon and select *Color_By* followed by *alignment.*
Default (0.5)

---

densityCutoff

The neglected fraction of the total atomic electron density in the course of calculation of the grid electron density from atomic positions. Atomic density distribution is approximated by two Gaussian functions which need to be truncated for computational efficiency. See also:

- `make map cell` command and related operations with the electron density,

Default (0.1)

---

dielConst

dielectric constant of the solute used in Coulomb, distance-dependent, MIMEL, and boundary element `electrostatic` calculations. If `electroMethod="distance dependent"` the actual dielectric constant is a product of `dielConst` and a distance from a change.
See also: `dielConstExtern`, `term "el"`
Default (4.0)

---

dielConstExtern

dielectric constant of the solvent exterior used in MIMEL and boundary element `electrostatic` calculations.
Default (78.5)

---

drop

expected initial function drop in local `minimization`. The parameter is used to evaluate initial step size. If your function is already very close to its minimum, it is a good idea to reduce the parameter, otherwise the procedure will start with an inappropriately large step.
Default (10.0)

---

fogStart

relative Z-depth with respect to the front clipping plane at which fogging starts. With this parameter you can keep some area in front without any fog and than start gradually increasing the effect until the back clipping plane.
To activate **fog** use **Ctrl-D** , or click on the FOG GUI button, or use the `display volume` command. Clipping planes can be moved with `Ctrl-MiddleMB` (front plane) and MiddleMB - left 5% margin (back plane). Actually the mapping of these operations to particular keystrokes is flexible and is defined in the `icm.clr` file. For Linux it is useful to redefine the back-clipping plane movements to

```
 mode   9  Right5-Mid                # Move rear clipping plane
```

Right5 means that you use the 5% right margin of your window.
Usually the fog color is the same as the background color. You can change the fog color with the
`color volume` *Color*
command. From the **command line** the fog can be switched on and off with the `display volume` and `undisplay volume` command

The value of `fogStart` is saved and restored as a system preference. If its value is negative, the fog is not activated, if the value is a positive number between 0. and 1., ICM sets the fog flag on.

Default (0.3)

---

gapExtension

Relative gap extension penalty used in an `alignment` procedure. The absolute gap penalty is calculated as a product of `gapExtension` and the average diagonal element of the `residue comparison table`
Default (0.15)
See also `gapFunction` , `Align` .

---

gapOpen

Relative gap opening penalty used in an `alignment` procedure. The absolute gap penalty is calculated as a product of `gapOpen` and the average diagonal element of the `residue comparison table` You may vary gapOpen between 1.8 and 2.8 to analyze dependence of your alignment on this parameter. Lower pairwise similarity may require somewhat lower gapOpen parameter. A value of 2.4 (gapExtension=0.15) was shown to be optimal for structural similarity recognition with the `Gonnet et. al.`) matrix, while a value of 2.0 was optimal for the `Blosum50`) matrix (`Abagyan and Batalov, 1997`).
Default (2.4).
See also `gapFunction` , `Align` .

gpWeights

the `rarray` of seven weighting factors for the `property grid` penalty term in the energy function.

Example:

```
gpWeights = Rarray( 7, 1. )  # seven weights of 1. each
```

See also:

- `set type property` # setting atom type contributions
- `make map potential` "gp" .. # generating up to seven grid maps.
- `term "gp"`

hbCutoff

(Angstroms) cutoff radius for `hydrogen bonding` interactions.
Default (3.0)

lineWidth

the real width of lines used to display the `wire` representation of chemical bonds. See also `IMAGE.lineWidth` parameter which controls line thickness in molecular images generated by the `write postscript` command, and the `PLOT.lineWidth` which controls the width for the `plot` command.
Default (1.0)
Example:

```
build string "se nad"  # NAD molecule
lineWidth = 3.
wireStyle="chemistry"
display
```

See also: `GRAPHICS.grobDotSize` , `GRAPHICS.grobLineWidth` , `GRAPHICS.mapLineWidth` , `IMAGE.lineWidth` , `PLOT.lineWidth` , `PLOT.gridLineWidth` .

listUpdateThreshold

the real maximal displacement of an atom during local `minimization` which triggers the recalculation of the interaction lists. This mechanism can be suppressed by setting the `l_updateLists` variable to `no`

See also: `l_updateLists` , `minimize` .

Default (1.5)

mapSigmaLevel

(in Rmsd values over the mean value). Margin value used for making `graphical objects` contouring the 3D density `map` .

See also: `map`
Default (1.5)

mapAtomMargin

Margin in Angstroms around selected atoms. The margin is added to the positional boundaries to define a submap index box in the `Map` (*map_source* , *as_*) function.

Default (3.0)

mcBell

average relative size of normally distributed `montecarlo` step from the center of an ellipsoid surrounding the multidimensional `variable restraint` zone.
Example:

```
 mcBell = 1.0        # places one standard deviation at the rs border
 mcBell = 2.0        # distribution is two times broader etc.
```

Default (1.0)

mcJump

maximum value (in degrees) of random angular distortion per variable during a stack *action* (as opposed to `mcStep` that is a part of a regular random step). The `mcJump` local random perturbation occurs if `visitsAction`, `highEnergy` or `rejectAction` ICM-shell variables are set to `"random"` . Randomization is a possible action in three problematic situations in `montecarlo` procedure.
Default (30.0)

mcShake

amplitude [Angstrom] of Brownian type the `montecarlo` random move applied to a molecule when one of the 6 variables defining its relative position is picked. Usually these variables may be selected by `v_myMolecule//?vt*selection.` The center of mass of the molecule randomly moves in an xyz sphere of **mcShake** radius. The molecule is also randomly rotated around a random axis with an amplitude equal to *mcShake* divided by the *MolecularRadius* . This parameter is also used as a default amplitude for the `randomize` command where the six position/orientation variables are selected.
Default (2.0)

mcStep

`montecarlo` step size (degrees). Maximum random change of one variable. This parameter is also used as the default amplitude for the `randomize` command
Default (180.0)

mfWeight

the overall weighting factor for the `"mf"` penalty term. This term may contain any user-defined energy or penalty function depending on pairs of atom types and interatomic distances. The parameters for the term are stored in the `icm.pmf` file and loaded with the `read pmf` *s_pmfFile* command.
The weighting factor will determine the "mf" term contribution with respect to the energy terms.
See also: `"mf"` term, `cnWeight`, `dcWeight`, `rsWeight`, `ssWeight`, `tzWeight`, `ssWeight`, `gpWeights` .

Default (1.0)

mimelDepth

The fraction of an estimated molecular radius which is taken as a radius of the probe sphere used by the `MIMEL` algorithm. The accessible surface of this probe sphere is used to calculate the distance between a charge and the effective dielectric boundary. Described in detail on p. 991-992 of (`ato94{ Abagyan and Totrov, 1994}). For small molecules `mimelDepth = 0.3` is recommended.
See also:

- `mimelMolDensity`
- `electroMethod`
- `show area`

Default (0.5).

mimelMolDensity

a coefficient used to calculate the effective molecular radius from a number of atoms. Recommendation: do not touch it, unless you are an advanced user. See also the description of the `MIMEL` method.

See also:

- `mimelDepth`
- `electroMethod`

Default (1.0).

r_out

a real variable where some commands and functions (e.g. `show area`, `show volume`, `superimpose`, `minimize tether`, `Corr` , `Axis` , `Align`) store their output. Also, in the electrostatic calculations with the `MIMEL` or `REBEL` method, the solvation energy part of the electrostatic energy is returned in `r_out`.
Default (0.0). See also: `r_2out`.

r_2out

a real variable where some commands and functions (e.g. `Axis`) store their output.

Some r_2out outputs:

- `Align` or `align` : for 2 sequences: percent sequence alignment identity
- `Axis`(R_12transform) : helix rise in
- Energy(rs simple) : rmsd of normalized residue energies
- convert : maximal positional deviation upon conversion

Default (0.0). See also: `r_out` .

resLabelShift

is the translation towards the viewer (normal to the graphics screen) used to display a label in front of `cpk`'s or `skin`'s rather than bury the label under them. The recommended value is 4. See also: `resLabelStyle`
Default (0.0) to be used with more popular wire representation.

rsWeight

weighting factor for the `multidimensional variable restraints` penalty term.
Default (1.0).

---

selectMinGrad

default minimal gradient vector length for gradient atom selection (`a_//G`). This parameter is also used by the `montecarlo fast` command, which requires a value of 2. to 10. for optimal performance.
Example:

```
 read pdb "1fox"
 convertObject a_ yes no yes no
 show energy
 selectMinGrad=80.
 show a_//G
 display
 display a_//G cpk
```

Default (1.5).

---

selectSphereRadius

default sphere radius (in Angstroms) for atom selections in `Sphere( )` function, as well as the Gaussian 3D averaging radius in the `color ribbon` command with `ribbonColorStyle="reliability"`.
This parameter is also used in the `compare surface` command.
Default (5.0).

---

sfWeight

the overall weighting factor for the `surface` solvation energy term. If `surfaceMethod` is `"constant tension"` it can also be controlled by the `surfaceTension` parameter.
Default (1.0).

---

shininess

parameter defining the shininess of solid surfaces such as `cpk`, `ribbon`, `ball`, `stick`, `xstick`, and `skin` when they are `display`ed. Only values in the range [0.,128.] are accepted.
Example in which we generate a high quality CPK image:

```
build string "ASDW"
GRAPHICS.quality = 15.
shininess = 100.
display cpk
```

Default (20.0). Range: from 0. to 128.

---

ssThreshold

threshold distance between two Sg atoms of cysteine residues. This distance controls the automatic formation of disulfide bonds in some commands (e.g. `read pdb`).
Default (2.35).

---

ssWeight

weighting factor for the disulfide bridge (`"ss"`) penalty term.
Default (1.0).

---

ssearchStep

angular increment (in degrees) for variables in the systematic search (`ssearch` command) in so called "local" mode when the search is performed around the current conformation.
Default (30.0).

surfaceTension

surface energy density in kcal/mole/A^2. The surface energy which is a product of this parameter by the total solvent accessible area will be stored in the `"sf"` term, if `surfaceMethod` preference is set to `"constant tension"`.
Note, that if a part of the system is represented with grid potentials, one needs a special `m_ga` grid map for correct calculations of the surface accessibilities.
Default (0.012)

tempLocal

`montecarlo` simulation temperature for local deformation random moves. This temperature can be set higher than the normal `temperature` since a local deformation includes a larger number of variables and may require a higher temperature for efficient sampling. To set the same simulation temperature, specify:
`tempLocal=temperature`
in your script.
Default (5000.).

temperature

`montecarlo` simulation temperature. A new trial conformation with a higher energy than the current one is accepted with the probability of exp(-(Etrial - Enew)/RT)). RT is 0.6 kcal/mole for T = 300 Kelvin.
The effect of temperature on the `montecarlo` procedure is the following:

- to find the global minimum successfully one needs a combination of persistence if a chosen place with a good sense of when to stop searching in this place and move along to the next one.
- if the temperature is too high, the acceptance ratio improves (gets higher) and wider sampling becomes easier since more high energy conformations are accepted. The downside of this is the low "persistence" (or "lack of patience") of the search procedure. Instead of spending more time in each conformational `vicinity` to find the real global minimum, the procedure just tries a couple of sub-optimal conformations and jumps away.
- if the temperature is too low the procedure may not cover the global conformational space of interest.

* tempCycle (need to create a variable with that name) If you want to set a cooling temperature schedule, or even a periodic PCR-like temperature schedule, you can define a new rarray array called `tempCycle` which contains { *tempMax*, *tempMin*, *period*, *phase* } . For a simulated annealing schedule, set the period to be twice the value of the `mncallsMC` parameter. $T_{curr} = 0.5(T_1 + T_{2)} + 0.5\ \Delta T \cos(2\quad ($ *phase* + x / *period*)), x = [0:1]

```
tempCycle = {10000.,600.,2.,0.}  # the whole period is 2 times longer than the simulation
# it will start from t=10000. finish at about 600.
montecarlo
```

Default (300.).

timeLimit

the `real` running time of ICM in hours (wall time) before a long simulation is interrupted. This limit can interrupt a `montecarlo` procedure (another exit condition is `mncallsMC`),
Example:

```
timeLimit = 2.  # two hours
montecarlo    # will interrupt the simulation in 2 hours
```

Default (99999.0).

---

tolGrad

gradient tolerance criterion for local `minimization`. Minimization is stopped if the gradient root-mean-square deviation from zero is less than the parameter value.

See also: `tolFunc` , `mncalls` , `minNumGrad` , `l_updateLists` , listUpdateThreshold .
Default (0.05).

---

tolFunc

exit criterion for local `minimization` by looking at the amplitude of the function value decrease during the last four steps of minimization. Minimization is stopped if the function does not decrease more than the parameter value during the last four steps. The negative value means that this criterion is ignored. All criteria, namely `tolGrad`, `mncalls` and `tolFunc` , are checked simultaneously.

See also: `tolGrad` , `mncalls` , `minNumGrad` , `l_updateLists` , listUpdateThreshold .
Default (-0.01).

---

tzWeight

the overall weighting factor for the `tether` penalty term. You may need to increase it while minimizing a highly energetically strained molecule resulting from the initial steps of the `conversion` or `regularization` procedure. Additional atom specific weights can be introduced through atomic bfactors with `tzMethod="weighted"`

See also: `TOOLS.tsWeight` , `term ts`
Default (1.0).

---

vicinity

maximum angular root-mean-square deviation per variable (degrees) or cartesian root-mean-square deviation per atom (Angstroms) when two structures are still considered belonging to the same conformational family in `conf`ormational `stack` manipulations. The type of comparison is defined by the `compare` command.
Examples:

```
 compare a_//ca,c,n  # compare by Cartesian RMSD
 vicinity = 3.0      # conf. are similar if RMSD< 3 A

 compare v_//phi,psi # compare by angular RMSD
 vicinity = 40.0     # conf. are similar if aRMSD < 40 deg
```

Default (15.0) . Do not forget to set it to a lower value if Cartesian RMSD is `compare`d.

---

vwCutoff

(Angstroms) cutoff radius for `van der Waals` interactions and Coulomb `electrostatics` .
Default (7.5).

---

vwExpand

radius of a probe sphere used to `display` a dotted `surface` of a molecule. All van der Waals radii are expanded by this value. *vwExpand=0* corresponds to the `CPK` surface, *vwExpand=1.4* corresponds to the water-accessible surface. Be aware of the difference between the `waterRadius` , `vwExpand` and `GRAPHICS.surfaceProbeRadius` parameters: The `waterRadius` parameter is used in

- `show energy "sf"`
- `show [area|volume] skin`
- `display skin`

while `vwExpand` is used in

- `show [area|volume] surface`
- `Xyz (` *as_ r_distance* `surface)` # sampling points above the surface

and `GRAPHICS.surfaceProbeRadius` is used in

`display surface`
Default (1.4).

vwExpandDisplay

Obsolete. Replaced by `GRAPHICS.surfaceProbeRadius` . See also: `GRAPHICS.surfaceDotDensity`, `GRAPHICS.surfaceDotSize`

vwSoftMaxEnergy

Parameter defining maximal energy value of van der Waals repulsion at r -> 0. for the finite approximation van der Waals function (`vwMethod` = `"soft"`). This parameter must be greater than 0. kcal/mole. Note that in the "soft" mode, the electrostatic energy will be automatically buffered to avoid singularities. You will see that the electrostatic term `"el"` changes upon switching from `vwMethod=1` to `vwMethod=2` .
Default (7.0).

waterRadius

radius of water sphere which is used to calculate an analytical molecular surface (referred to as `skin`) as well as the solvent-accessible `surface` (centers of water spheres). Because of the complexity of skin calculations, it is not recommended that one play's with this parameter (of course, you rushed to do exactly that). Be aware of the difference between the `waterRadius` and `vwExpand` parameters: `waterRadius` is used in

- `show energy "sf"`
- `show [area|volume] skin`
- `display skin`

while `vwExpand` is used in

- `display surface`
- `show [area|volume] surface`

Default (1.4).

wireBondSeparation

the distance between two parallel lines representing a chemical double bond if `wireStyle` = `"chemistry"`.
Default (0.15 Angstroms).

xrWeight

the overall weighting factor for the `structure factor correlation` penalty term. See also: `xrMethod`.
Default (1.0).

Logical variables

ICM-shell logical variables are the following.

---

l_antiAlias

if `yes`, invokes anti-aliasing for lines displayed in the graphics window. This feature is not supported on all the platforms.
Default (`no`).

---

l_autoLink

if `yes`, tries to link molecules and alignments/sequences automatically. In case of degeneracy, i.e. identical sequences exist with different names, a molecule can be linked to two different alignments containing its sequence etc., the autolink procedure chooses the first occurrence. Use the `link` command to impose links explicitly, and the `show link` command to see them. Links can be used by the following commands and functions:

- `superimpose`
- `Rmsd` and `Srmsd`
- `set tether` *`ali_`* `...`

Default (`yes`).

---

l_bpmc

if `yes`, use Biased Probability Monte Carlo moves in the `Monte Carlo` procedure. See `Abagyan and Totrov, 1994` for reference. **Important:** the probability zones are described in the `icm.rst` file and should be assigned to a peptide before the `montecarlo` command with the
`set vrestraint` a_/*
command.
Default (`yes`).

---

l_breakRibbon

if yes, break too the `ribbon` if the distance between the reference atoms is larger than
`GRAPHICS.ribbonGapDistance`

Default (`yes`).

---

l_bufferedOutput

if `no`, suppresses pagination in the output of ICM commands (including the `show` command). Useful in batch jobs.
Default (`yes`).

---

l_bug

if `yes`, print some debug information
Default (`no`).

---

l_caseSensitivity

active in most commands and functions using string comparisons.
Default (`no`).

---

l_commands

if no, do not show commands in batch mode
Default (`yes`).

---

l_confirm

if `no`, overwrite the contents of an existing file; ask permission to overwrite it otherwise.
Default (`no`).

---

l_easyRotate

allows faster handling of images in the graphics window. If `yes`, then the currently displayed solid representations (e.g., `ribbon`, `skin`, `cpk`, etc.) are temporarily hidden if an operation like rotation or translation is undertaken. Only the `wire` representation remains allowing quick manipulation with the object in use. The previous type of display is restored when rotation or translation is completed. The parameter can be toggled by a keystroke if you assign the `l_easyRotate = !l_easyRotate` with the `set key` command.
Default (`no`).

---

l_info

if `yes`, print info messages
The default value is `yes` .

---

l_minRedraw

if `no`, suppresses redrawing of a displayed structure at each `minimization` step. The new minimized structure will be redrawn only at the end of minimization. Useful when the graphics is slow or the structure is heavy.

l_neutralAcids

Several commands such as `read mol`, `read mol2`, `build smiles` and `set bond auto` include automated assignment of aromatic systems as well as some resonance structures in O-C=O, O-S=O, PO3, O-N=O, and NO3. The automated conversion invoked with the `l_readMolArom` variable set to `yes` reassigns the bonds in the group to be equivalent. For the acidic groups it leads to the *charged* form with two partial charges of -1/2 or -1/3. If you want to suppress this transformation for the CO2,SO2 and PO3 groups only set the `l_neutralAcids` flag to `yes` . In this case the acidic groups will be kept unchanged.
Example:

```
l_neutralAcids = yes
read mol s_icmhome+"ex_mol.mol"
wireStyle=2
display only a_  # the acidic group is uncharged
build hydrogen
```

Default (`no`).
See also: `l_readMolArom`,`read-mol{read mol}, `read mol2`, `build smiles` and `set bond auto`.

---

l_out

a logical variable similar to `i_out` and `r_out` .
Default (`yes`).

l_print

if `yes`, show `print` command with arguments as well as the result of its action.
Default (`no`).

l_racemicMC

Activate switching between stereoisomers at chiral centers during `montecarlo` . This flag can also be dynamically activated with the `chiral` option of the `montecarlo` command. To reset the chirality status of an atom use the `set chiral` command
Example:

```
build string "se nter his cter"
set chiral a_/his/ca 3  # set chirality flag to 3 (means a racemic mixture)
unfix V_//FC    # unfix phases for stereoisomeric rearrangements
compare a_//*
vicinity = 1.

l_racemicMC = yes
montecarlo v_//!?vt*   # will switch between stereoisomers

display
display atom label type=6   # to see the isomers
# now you can browse the stack solutions
```

l_readMolArom

if `yes`, automatically assigns aromatic rings and resonant structures (CO2,SO2,PO3,NO2,NO3) from patterns of single and double bonds upon `reading` objects, mol and mol2 files or build from smiles. The automated assignment module is also called by the `set bond auto` command.
If this flag is set to `no` , the `build hydrogen` command will have problems with resonant structures, such as carboxyl groups, - a hydrogen will be attached to the oxygen connected with a single bond to the carbon.
Example of a recommended best conversion procedure for chemical library files:

```
 l_readMolArom = yes # it is the default, but just in case
# you also want to use l_neutralAcids = yes
 read mol s_icmhome + "ex_mol"
 for i=1,Nof(object)
   build hydrogens   # may have problems if l_readMolArom = no
   set type mmff     # also improves the aromatic system assignment
   set charge mmff
   convert           # makes an ICM object
 endfor
```

Default (`yes`).
See also: `l_neutralAcids` which allows one to keep acidic groups unchanged and uncharged.

l_showAccessibility

show the residue accessibility string assigned to a `sequence` generated from a three dimensional structure in the commands `show sequence` , `show alignment`, `write alignment` . The relative residue accessible area is expressed by an integer number in a scale from 0 to 9 (0-fully buried, 9-fully exposed).
Example:

```
 read pdb "1crn"
 show surface area   # calculate atomic and residue accessibilities
 make sequence a_1   # generate a sequence
 l_showAccessibility=yes
```

```
show 1crn_m
```

Default (yes).

---

l_showMC

display one-line info about each Monte Carlo trial conformation.
Default (yes).

---

l_showMinSteps

display every step of the local minimization procedure.
Default (no).

---

l_showResCodeInSelection

if yes, shows one-letter code for amino-acid residues in residue selections, e.g. a_/^F12:^A23 instead of a_/12:23 . The amino-acid code is preceded by a *caret* symbol **^**. In versions older than 3.1 the amino-acid code was not shown.

l_showSpecialChar

if yes, displays unprintable characters with the show string and list string commands in text format (like \a \t \n). This flag does not apply to the print command.
Default (no).

---

l_showSites

show the site string assigned to a sequence in the commands show sequence, show alignment, write alignment. The one-letter site codes are given below.
Default (yes).

---

l_showSstructure

show the secondary structure string assigned to a sequence in the commands show sequence, show alignment, write alignment.
Default (no).

---

l_showWater

if yes, all water molecules are shown in the output of commands such as show molecule or show a_* . Set it to no to skip the usually long lists of water molecules in PDB structures.
Default (yes).

---

l_showTerms

Obsolete. Now you can achieve the same via s_icmPrompt variable.
Examples:

```
s_icmPrompt = "icm/%o/%e> "  # equivalent to l_showTerms=yes
```

---

l_updateLists

if yes, updates the atomic interaction lists during minimization once the maximal displacement reaches listUpdateThreshold . The interaction lists are lists of atom pairs involved in van der Waals, electrostatic or hydrogen bonding interactions and within the vwCutoff or hbCutoff distance. If this parameter is set to `no` , the lists are not recalculated. To trigger a recalculation, use the

```
delete list
```

command.

Default (`yes`).

l_warn

if `yes`, print warning messages. If you want to see warning messages (i.e. `l_warn = yes`), but suppress some of the messages, use the `s_skipMessages` variable (e.g. `s_skipMessages = "[147][148]" )`.
Default (`yes`).

---

l_wrapLine

wrap long lines if `yes`. If `no` truncate long lines and add a dollar sign ($) to indicate that truncation has occurred.
Default (`yes`).

---

l_writeStartObjMC

write the starting object in the `montecarlo` command to a file. This object will have the same fixation (set of free and fixed variables) as in your montecarlo simulation. In case the variable is set to `no`, the same object can be generated if you repeat the fix and unfix command as in your simulation script.
Default (`yes`).

---

l_xrUseHydrogen

defines whether hydrogen atoms are used in calculations of crystallographic structure factors from atom coordinates (the term).
Default (`yes`).

String variables

System string variables are predefined in the shell. New string variables can be created via assignments, e.g.

```
a = "What took you so long?"
txt = """
Once upon a time some evil dwarfs
filed a patent claiming the right to
paint shirts blue color
"""
```

, or the `read string` command or a function returning a string (e.g. `a = String(2.3)`) or created by a command as one of the output variables, eg `s_out`.

s_alignment_rainbow

This variable now controls how alignments are colored automatically by properties like conservation or entropy. For example:

```
s_alignment_rainbow = "pink/white/white/lightyellow/yellow/yellowgreen/green"
```

Note: this variable does not influence the consensus-based coloring via tables CONSENSUSCOLOR and CONSENSUS

s_blastdbDir

return directory with Blast-formatted sequence files for ICM sequence searches. By default the directory is set to the `$BLASTDB` system shell variable. The variable can also be explicitly defined in the `user_profile.icm` or `_startup` file. In order to start using the `$BLASTDB` shell variable, delete explicit assignment of the `s_blastdbDir` from your `_startup` file or add

```
s_blastdbDir=Getenv("BLASTDB")
```

to your `~/.icm/user_startup.icm` file.
The `find database` family of sequence/pattern search commands use the `s_blastdbDir` directory.

s_editor

a string to invoke an external editor.
Attention!!! Always use the call to the program which starts the program in the foreground. For example: use "jot -f" rather than just "jot", since the default is running in the background.
Examples:

```
 s_editor = "vi"       # good old vi, does not require a separate window
 s_editor = "jot -f"   # popular SGI editor
 s_editor = "xedit"    # simple and exists for X on every platform
 s_editor = "notepad" # exists for PCs
```

s_entryDelimiter

a string which delimits entries in the `database` output of a `table` or a set of arrays, generated by the `show database` or `write database` commands. The **%i** specification at the end will be replaced by the current number of the entry and carriage return.
Default: ("#______________________________ %i")
Example:

```
 s_entryDelimiter="//\n"  # EMBL-database delimiter
```

s_errorFormat

defines the exact appearance of the ICM error messages. Specification %s corresponds to the minimal ICM error message. If %s is missing all error messages are reduced to the specified text. If **s_errorFormat** is equal to the empty string (""), all error messages will be suppressed. If icm is started in the "web" mode (i.e. with the **-w** *path* flag), the variable is automatically set to `"<hr><h3>Error: %s</h3><hr>"` .
Examples:

```
 s_errorFormat=""              # do NOT print error messages
 s_errorFormat=" Error> %s"    # standard error messages
 s_errorFormat=" Erreur> %s"   # French version
                               # html-padding
 s_errorFormat="<hr><h3>%s</h3><hr>"
 s_errorFormat=" Fehler> der Betrieb ist verboten"
                               # replace all the messages by this text
```

s_fieldDelimiter

contains characters which are considered as field delimiters by the `Field` and `Split` functions, as well as by `read column` and `write table` commands. In "Split" and "read table" one can also specify the field delimiter explicitly.
Important. If a character is duplicated in *s_fieldDelimiter* (e.g. `s_fieldDelimiter="::"`), then multiple occurrences of this character will be ignored. Otherwise, EMPTY fields will be created between each pair of identical delimiter characters.
In `write table` s_fieldDelimiter is honored only if is a one-letter symbol, like "," or "\t".
See also the opposite operation, merging members of string array into one string: `Sum(` *S_, s_separator* `)`
Examples:

```
 s_fieldDelimiter="\t"   # "aaa\t\t bbb" splits into "aaa",""," bbb"
 s_fieldDelimiter="\t\t" # "aaa\t\t bbb" splits into "aaa"," bbb"
```

Default (`"  \t\t"` i.e. two blanks, two tabs, meaning skip multiple blanks or tabs). Another reasonable possibility is `"  \t\t\n\n"` which means skip blanks,tabs and carriage returns.

s_helpEngine

path to the HTML help file browser program. If you have no HTML browser, the default setting is `s_helpEngine="icm",` so you can use the simple internal ascii help-file viewer `more` filter ('q' - to stop, '/' to find a string, 'Enter' - next screen). If the desired help information is not found, just type help and then use '/' plus the search pattern to perform the context search in the whole help file.
Examples:

```
s_helpEngine="/usr/bin/netscape"
s_helpEngine="mozilla"  # make sure you can start it in the UNIX shell
s_helpEngine="icm"      # why would one need more?
```

s_icmhome

defines the home directory of the ICM program. This directory contains all standard ICM databases, all scripts, examples, documentation, initial configuration files (later users can override them with the files stored in the `s_userDir` directory.
The Linux icm-rpm package creates `s_icmhome` in `/usr/icm` directory.

s_inxDir

defines directory from which icm - index files for large sequence or chemical databases are stored. This variable is used by the `write index` command. By default `s_inxDir` is set to `s_icmhome` + `"/data/inx/"`.
See also: `read index` , `write index` .

s_icmPrompt

defines the ICM-prompt string. This string contains text and a bunch of wild cards for:

- **%o** - name of the current molecular **object**
- **%e** - list of the active **energy** terms (see the `set terms` command)
- **%t** - **time** spent in ICM (may be convenient for scripts)
- **%T** - astronomical **Time**
- **%%** - **%** character
- **%#** - icm-command order number

Be smart, see the `energy` or penalty `terms` you are using by adding **%e** to the prompt string.
Examples:

```
s_icmPrompt="%## "                    # for minimalists
s_icmPrompt=""                        # for super-minimalists
s_icmPrompt="%T> "                    # for anxious paranoiac freaks
s_icmPrompt="MY_ICM/%o/%e/%T/%#> "    # for the verbose
s_icmPrompt="Hi-hi|%e-^%o+%T>    "    # for the messy
s_icmPrompt="Icm command number %#> " # for the retarded
s_icmPrompt="Hey dude, type something" # for dudes
s_icmPrompt="%o/%e> "                 # for humble and wise researches
```

Default: `"icm/%o> "`

s_imageViewer

defines the command to view the image files (`tiff`, `png`, `targa` and `rgb` formats) if the `display` option is specified. An alternative to the default is the `"xv"` program. See also the `write image` command.
Default for SGIs (`"imgview")`.

s_javaCodeBase

path to the folder containing java applet class files. Java applets are currently used by the `web` or `write html` commands with a chemical table.
Default: `"/Java/"`

s_labelHeader

defines a prefix string for all labels. For example, when displaying CPK atoms you may move the label to the right of the atom center by

```
 s_labelHeader="     "
```

Default (`""` - an empty string).

s_lib

ICM library name root. If you redefine it to say "new", ICM will start to look for the following library files: new.cod, new.bbt, new.bbs, ... etc. in the $ICMHOME directory.
Default (`"icm"`) .

s_logDir

when you quit an icm-session, a `_seslog.icm` file is automatically stored. If the `s_logDir` variable is empty, it is stored to the `s_userDir` + `"/log/"` directory. However one can redirect it to the current working directory (`"."`) or any other directory.
The same logic applies to the `_crashlog.icm` file which is created when ICM crashes.
Examples:

```
s_logDir = "." # _seslog.icm stored in the current working directory
s_logDir = ""  # to the current working directory
```

s_out

a string where some commands store their string/text output. See also: `printf` `read database` `read string`, `read table`, and `read unix`,
Default (`"is where the string/text is stored"`).

s_pdbDir

directory containing the PDB database of 3D structures and defining the location of the pdb files for the `pdbDirStyle` from 1 to 5 (the ftp and http styles are controlled by a different variable). These files can also be easily downloaded directly from the PDB site if the variables are set as in the example below. PDB distributions can exist in several styles (all files in the same directory, or divided etc.). The style is defined by the `pdbDirStyle` preference.
The pdb directory also contains the `derived_data` subdirectory with useful files (pdb sequences, index files etc.)

Attention! Note that if the `pdbDirStyle` is set to 6 or 7 this variable is NOT USED. Variables `s_pdbDirFtp` and `s_pdbDirWeb` are used instead! (it was hard coded until version 3.5-2)
Example:

```
 s_pdbDir ="ftp://ftp.rcsb.org/pub/pdb/data/structures/divided/pdb/"
 pdbDirStyle = "ab/pdb1abc.ent.Z"

 s_pdbDir = "/data/pdb/"
 read sarray s_pdbDir+"/derived_data/index/source.idx"
 source = Tolower(Trim(Field(source,1)))
 for i=1,Nof(source)
   read pdb source[i]
#  do some analysis
   delete a_*.
 endfor
```

Default (`"/data/pdb/")`. It is usually redefined in the _startup file.

---

s_pdbDirFtp

If `pdbDirStyle` is set to 6 (for the ftp access), the location of the ftp site is controlled by this variable. Example:

```
pdbDirStyle=6
s_pdbDirFtp = "ftp://ftp.rcsb.org/pub/pdb/data/structures/divided/pdb/" # an old site
read pdb "1crn"
```

s_pdbDirWeb

If `pdbDirStyle` is set to 7 (for the http access), the location of the web site and the request format is controlled by this variable. Example:

```
pdbDirStyle=7
s_pdbDirWeb = "http://www.rcsb.org/pdb/files/%s.pdb.gz" # an old web site
read pdb "1crn"
```

s_projectDir

a **relative** path to the directory in which icm-projects (all the icm-objects in a session) are stored. This path is appended to the `s_userDir` directory.

s_printCommand

a command to print text or postscript files. This command is invoked if the `print` option is specified in the `write image postscript` or `write postscript` commands. Customize this string. Default (`"lp -c"`).
Example:

```
 s_printCommand = "lp -c -d ColorPrn22"
 write image postscript print  # save image and print
```

---

s_prositeDat

is a file containing the full file name of the `prosite` database of protein patterns. This file is not large and is distributed with ICM. If you have your own copy of prosite, redefine the variable and delete prosite.dat in the $ICMHOME directory to avoid redundancy.
Default (`"prosite.dat"`). It is usually redefined to `s_icmhome+"prosite.dat"` in the `_startup` file.

---

s_psViewer

a PostScript viewer used while you are in ICM session. A command to invoke is to be:

```
 unix $s_psViewer </tt><i>your PostScript file name</i>
```

Default is system specific.

---

s_reslib

name of the `icm residue library` that contains discriptions of chemical residues. The file will be loaded from the $ICMHOME directory.

To create a new residue use the `write library` command.
Default (`"icm"`) .

s_skipMessages : ignore specific error messages

In ICM all error and warning messages are numbered (e.g. `" Warning> [123] .. "`). You may specify a set of message numbers which you want to suppress. While the messages are suppressed the error code can still be returned with the `Error`(number) function.
Example:

```
 a = 1
 if = 2       # deliberately generate error
  Error> [2073] illegal IF: wrong condition in  if=2

 s_skipMessages = "[2073]"
 if = 2        # now no message is generated
 if Error(number)==2073  quit

 a = yes      # generates another error
  Error> [696] wrong assignment or name conflict
 s_skipMessages = "[2073][696]"
 a = yes      # hides the error message

 234*2352352532
 Warning> [147] number 2352352532 is too big for an integer (>2147483647)
 0
 s_skipMessages = "[2073][696][147]"   # suppress the warning
 234*2352352532
 0
```

See also: `errorAction` , `s_errorFormat` .
Default (`"[3000][3012]"` just to show an example).

s_sysCp

automatically filled out `string` containing the **copy** file command for the current operating system. It is advised to use this variable in scripts for cross-platform portability.

s_sysLs and s_sysLtt

automatically filled out `string` containing the **list** file command (or list files sorted by modification time) for the current operating system. It is advised to use this variable in scripts for cross-platform portability.
Example:

```
sys $s_sysLs   # to list files in the current directory
sys $s_sysLtt   # to list files in the current directory
```

s_sysMv

automatically filled out `string` containing the **move** or **rename** file command for the current operating system. It is advised to use this variable in scripts for cross-platform portability.

s_sysRm

automatically filled out `string` containing the **delete** or **remove** file command for the current operating system. It is advised to use this variable in scripts for cross-platform portability.

s_tempDir

scratch directory for temporary files (some montecarlo files will be saved there).
Default (`"/usr/tmp/"`) .

s_translateString

a set of characters used in the ascii representation of numerical values of arrays, matrices and maps. See also the `String` function and the `show map` command.
Default (`".:*0#"`).

s_userDir

The path to the user directory containing ICM-related and ICM-generated data files.
The suggested _startup file sets this variable to a subdirectory `.icm` of the user $HOME directory (`$USERPROFILE` for Windows), but you may set it anywhere you want.
Default (`"$HOME/.icm/"`).

s_usrlib (obsolete)

an obsolete variable. The new mechanism to add new `icm residue libraries` uses the `LIBRARY.res` sarray. You can generate the entries using the `write library` command.
Default (`"usr"`).

s_webEntrezLink

defines the NCBI Entrez link.
See also: `webEntrezOption`, Default (
`"http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&form=6&uid=%s&Dopt=`

s_webViewer

An obsolete variable. Web browser is defined by OS default.

s_xpdbDir

path to the ICM XPDB database root of compact binary ICM objects which are annotated with the site information. The root directory contains pdb-style subdirectories with named after the 2nd and 3rd character of the four-letter code. The advantage of the XPDB database is the speed of reading and smaller size than PDB. XPDB entries are read about hundred times faster!

Here we compare the execution times for the pdb and xpdb files:

```
eos:/home/ruben/icm> time ./icm -s -e 'read object "/data/xpdb/1ffk.ob"'
0.450u 0.090s 0:00.54 100.0%
eos:/home/ruben/icm> time ./icm -s -e 'read pdb "/data/pdb/ff/pdb1ffk.ent.Z"'
38.800u 0.430s 0:42.11 93.1%
```

An xpdb directory can be in a remote location, e.g.

```
s_xpdbDir = "http://ablab.ucsd.edu/xpdb/"
read binary pdb "1crn"
```

Preferences

Preferences inside the shell are multiple choices (the outside persistent parameters are in ~/.config/Molsoft.conf , see `preference system`) . You can `show` and `list` them. You can change a preference by assigning it to:

- the item number
- the item name
- **"nextItem"** string
- 0 (the same as **"nextItem"**)

Examples:

```
 resLabelStyle = 3              # 3-rd choice
 resLabelStyle = "Ala 5"        # assign by string
 resLabelStyle = "nextItem"     # go to the next item in the list
```

Preferences are *temporarily redefined* just for one `command` , if specified after the command, e.g.

```
minimize v_//x* electroMethod=2
```

The ICM preferences can be divided into two groups:

- **persistent**, - the ones which user can modify and write with the `write system preference` *parName* command. The location of the resulting file depends on the Operating System:
  - ♦ Unix: ~/.config/Molsoft.conf
  - ♦ Mac: /Library/Preferences/com.molsoft.plist
  - ♦ Windows: windows registry

  Example: `GRAPHICS.chainBreakLabelDisplay`
- **non-persistent:** if you need to change them inside a script or in your `user_startup` file.

The **persistent** parameters can be found in the GUI menu under Preferences. The ones which are not on that menu, are non-persistent.

See also: `preference system` for the location of the file with modified user preferences and commands/ways to change them

Persistent Preferences

The `preferences` can be divided into persistent and non-persistent as described above. The non-persistent (like `TOOLS.tsToleranceRadius`) need to be changed in a macro or script when needed. The persistent ones can be searched and changed from the interface (the **File.Preferences** menu) or directly in a file between the ICM sessions for Unix and Mac. From the command line one can change the preference and issue the `write system preference` *prefName* command.

User preferences in Linux. The user preferences (a subset of all preferences that can be modified in ICM) can be modified by the user from the GUI (Menu File/Preferences). The modified preferences are then stored in the `~/.config/Molsoft.conf` file. You can modify those preferences manually provided there are no open ICM sessions. Also, if you want to restore the defaults, simply delete the lines in question in `Molsoft.conf.`

User preferences on a Mac.

```
open /Library/Preferences/com.molsoft.plist  # or
rm /Library/Preferences/com.molsoft.plist  # or
```

See also: `FAQ`

User preferences on a Windows box:are stored in registry and can-not be easily viewed with a text editor (need registry viewer).

See also: write-system-preference command.

accessMethod

Defines if the `show area surface` command calculates absolute or relative solvent accessible area for each atom. This area can be stored as absolute value in square Angstroms or relative value from 0. to 1. and can be returned by the `Area(` *as_* `)` function. Note that this preference does not work for residues. To calculate the relative residue area use the `Area( a_/* )/Area( a_/* type )` ratio of functions.

1. "absolute surface" "relative surface"
2. "multByAccMap"

Example:

```
show area surface a_1 a_1  accessMethod = 2 waterRadius=1.4
color a_1//* Area( a_1//* ) # returns numbers from 0. to 1.
```

The third method ("multByAccMap") uses `m_ga` map in a straightforward manner, just multiplies the atomic accessibilities by the map value. `m_ga` map is supposed to have values from 0. to 1.

alignMethod

alignment method used in the `Align` and `Score` functions and `find database` command (as described in `Batalov and Abagyan, 1999`).

1. "ZEGA"
2. "H-align" <- the best choice
3. "frame-H-align" # align DNA sequence against protein sequence or protein sequence database

See also:

- `gapFunction`,
- `accFunction`,
- `alignMinCoverage` (0.5) - minimal ratio of the aligned residues with respect to the shorter sequence length.

atomLabelStyle

style of atom labels invoked by clicking on an atom or the `display atom label as_` command. You may display name, electric charge (q) and/or `mmff` atom type. Options are the following:

1. `"cb1"` <== default
2. `"cb1 q"` (atomic charge)
3. `"cb1:FC"` (formal charge and chirality)
4. `"cb1 all"` (different atomic properties)
5. `"cb1 mmff q"`
6. `"C"` (chemical atom name for non-H and non-C atoms, formal charge and chirality)
7. `"[C]"` (chem. name, formal charge and chirality on a rectangle)

The last two choices use periodic table convention to label atoms, and the label is positioned into the center of atom. In the latter case ("[C]") a rectangle of the background color is used to highlight the label. Be careful since in the latter case the selection mark (green cross) is hidden.
Examples:

```
 build string "se his"
 atomLabelStyle = "[C]"
 wireStyle = "chemistry"
 lineWidth = 3.
 display atom label wire black # press Ctrl-A
 color background white
 write postscript "tm"         # save the results
#
 atomLabelStyle = "C"
 display xstick
 set type mmff                 # press Ctrl-A again
```

atomSingleStyle

display style of isolated atoms in the `wire` mode.

1. "tetrahedron"
2. "cross"
3. "dot"

The size of the first two representation is controlled by the `GRAPHICS.ballRadius` parameter and the line width (especially important for the "dot" style) is controlled by the `lineWidth` parameter.

cnMethodAverage

method of calculating an effective distance for NOEs between groups of protons This multi-center NOEs can be set with the

```
set drestraint all as_group1 as_group2 i_Type
```

command. Two methods are available.

```
 cnMethodAverage  = "R6"
        1 = "R6" <-- current choice
        2 = "nR6"
```

The first mode calculates effective distance `r` as `r=(1/N Sum(r^-6))^-1/6`, where `N` is `n1*n2.`The second mode calculates effective distance as `r=(Sum(r-6))^-1/6` Depending on the number of protons in each group, the difference in the effective distance may differ from 12 to 34%.

compareMethod

This method is usually set by the `compare` command. The last two methods perform chemical equivalency matching.

```
 compareMethod    = "variables"
        1 = "variables" <-- current choice
        2 = "atoms static"
        3 = "atoms superimposed"
        4 = "atoms interface"
        5 = "chemical static"
        6 = "chemical superimposed"
```

All methods except the first one ("variables") will use atom selection. Example:

```
  compare a_LIG.
  compareMethod = 5
  montecarlo
```

dcMethod

defines the algorithm for the `density correlation` calculation which is the correlation between the static `density distribution` and a virtual map generated from atomic positions on the fly.

1. `"exact"` <- default
2. `"unnormalized"`

Explanation:

1. The `"exact"` density correlation penalty function uses the Pearson's correlation coefficient. The correlation coefficient is then shifted by +1 so that the function ranges from `0.` to `2.` rather than from `1.` to `-1.` ***DC = 1 - Sum(*** $\boldsymbol{D_i}$ ***- < D >)(*** $\boldsymbol{A_i}$ ***- < A >)/(N * Rmsd(D)*Rmsd(A))*** The term has analytical derivatives with respect to the internal coordinates and can be efficiently locally `minimized`. This term requires additional memory allocation equal to the current map size and is two times slower than the unnormalized term.
2. The "unnormalized" density correlation. Formula: ***DC = 1 - Sum(*** $\boldsymbol{D_i}$ ***- < D >)(*** $\boldsymbol{A_i}$ ***- < A >)/ N*** where $\boldsymbol{D_i}$ is a map value in point *i* , and $\boldsymbol{A_i}$ represents the density generated dynamically from atomic positions. The differences from the `"exact"` term are the following:
    - ♦ scaling is arbitrary in contrast to `"exact"` term. Therefore you have to estimate a reasonable `dcWeight` value if `"dc"` is optimized along with the other energy or penalty terms.
    - ♦ The `"unnormalized"` term does not require additional memory and is two times faster than the `"exact"` term. The term has analytical derivatives with respect to the internal coordinates and can be efficiently locally `minimize`d.

electroMethod

defines method used for the electrostatic energy evaluation. Four options are available:

1. "Coulomb"
2. "distance dependent" <- default
3. "MIMEL"
4. "boundary element"

The meaning:

1. The Coulomb `electrostatics` is defined as $\boldsymbol{U = q_1 * q_2 / D * r_{12}}$ with $\boldsymbol{D}$ = `dielConst` .
2. In the distance-dependent dielectric model D in the above formula is set to `dielConst`*r, where `r` is an interatomic distance.
3. The "MIMEL" electrostatics allows one to evaluate the free energy of a molecule in water environment by the `Modified IMage ELectrostatics` approximation at every iteration of the `Monte Carlo`, or `search` procedure. This energy will only be calculated for a static structure or at the end of local minimization (so called "double energy scheme", see `Abagyan and Totrov, 1994` section (e) on p.992, or `Abagyan, Totrov and Kuznetsov, 1994` p. 10, for reference).). The MIMEL energy consists of the Coulomb energy, which is calculated for all the atom pairs at the current `dielConst` value, and the electrostatic solvation energy which is a sum of "selfEnergy" and "crossEnergy" and is returned in the `r_out` real variable upon completion of the calculation in the `show energy` command. A more accurate evaluation of the electrostatic solvation energy can be obtained with the `boundary element` method.
4. The `boundary element` method provides an accurate solution of the Poisson equation. The dielectric boundary is defined by the accurate `analytical molecular surface` (skin) and all the local charges stay exactly where they are. The boundary element method does not rely on any 3D grid and is free from dependence on the grid size. The ICM implementation of the boundary element method is fast and accurate. During the local `minimization` the derivatives with respect to the internal coordinates are not calculated (similar to the MIMEL method). The distance dependent dielectric model is used during minimization instead. At the end of the local minimization the electrostatic energy is replaced by the more rigorous `boundary element` energy.

errorAction

action taken after an error has occurred.

1. = `"none"` # error flag is set (see the Error() function)
2. = `"break"` <- default # exit from loops and macros
3. = `"exit"` # exit from a script into shell
4. = `"quit"` # quit ICM: useful for CGIs

Specific error messages can be suppressed with the `s_skipMessages` (e.g. `s_skipMessages = "[696][2073]"`)
See also: `s_errorFormat`, `interruptAction`

exitSessionStyle

Together with `s_logDir` controls where and what session files are saved upon exit/quit from ICM.

`exitSeslogStyle` = `"full seslog"` by default.

1. = "none" # no files are saved
2. = "full seslog" = "user session" # only user commands saved to session.icm
3. = "both" # both files are saved

This parameter can be redefined and saved to user preferences.

ffMethod

force field used in the `show energy`, `minimize`, and `montecarlo` commands.

1. = `"ecepp"` <- default
2. = `"mmff"`
3. = "icff" an experimental force field obtained by re-parametrization of the mmff force field into the internal coordinate space and derivation of the parameters specific for a particular covalent geometry.
4. = "icmff" a new forcefiled (Arnautova,Abagyan,Totrov, Development of a new physics-based internal coordinate mechanics force field and its application to protein loop modeling. Proteins. 2011 Feb;79(2):477-98.). To activate it run the set_icmffmacro.

Note that `minimize cartesian` temporarily enforces ffMethod = "mmff", since the ecepp force field is not applicable to the cartesian minimization.
To use the force fields you need to do the following:

- "ecepp"
    - ♦ `read library` (if it is not included in your _startup.icm file)
    - ♦ modify terms with the `set terms` command.
    - ♦ use `show energy` , `minimize`, or `montecarlo.`
- "mmff" in cartesian space (free covalent geometry). The command requires at least the `"`vw,af,bb,bs" terms and needs correct atom types and charges.
    - ♦ `read library mmff`
    - ♦ assign atom types: `set type mmff a_` . This operation requires correct
- chemical structure (when you build the molecule, make sure it is complete),
- bond types (check graphically with wireMethod=2, and change with the `set bond type` command), and
- formal charges (check graphically with the `atomLabelStyle=3`, and assign with the `set charge formal ..` command).
    - ♦ assign charges: `set charge mmff a_`
    - ♦ modify terms with the `set terms` command. The full set is: `set terms "vw,el,to,af,bb,bs"`
    - ♦ use `show energy` , `minimize`, or `montecarlo.`
- "mmff" in the internal coordinate space according to the current fixation. The use of the mmff force field is not recommended.
- "icmff". This new force field is designed to be used with the fixed covalent geometry and is faster than both mmff-cartesian and "ecepp". The icmff force field is still experimental and should be used with caution. The vacuum part of icmff requires only three terms: "vw,to,el". The solvation terms "sf,en" can be added. Icmff calculates parameters on the fly for a particular geometry. To use this force field use the following procedures:
- assign mmff types and charges, and load the mmff libraries (see above)
- to generate the starting conformation, minimize your molecule with `ffMethod = 2` and `minimize cartesian "14,to,bb,bs,af"` .
- set ffMethod to 3 and `set terms ""vw,to,el,sf,en" only` .
- use show energy or montecarlo

gcMethod

method defining how the `m_gc` map is used in the `"gc"` grid energy calculation. The `"gc"` method allows one to calculate interactions of a molecule with grid energy field representing another molecule (the first method), or treat the `m_gc` map as the electron density map. To see individual atomic contribution, use `show energy atom` command which places individual energies in the bfactorfield with a 20 unit offset.

1. `"vw"` <- default choice: current object interacts with the van der Waals field. Positive values repel, negative attract; Contribution from one non-hydrogen atom is `Eatom = 1.*Egc`
2. `"density"` : treats the `m_gc` map as positive electron density and pulls the object into it. The contributions of atoms are proportional to atomic number (the number of electrons), hydrogens are ignored: `Eatom = -AtomicNumber*Egc`
3. `"field"` : uses user-defined atomic `field` value, which can be set by the `set field` command and extracted with the `Field` (as_) command, as the relative weight of each atom. Anticipates that van der Waals type of the map (attractive negative values, repulsive positive) as in the first method. `Eatom = Field(atom)*Egc`

highEnergyAction

action taken upon achievement of the maximal allowed number of `montecarlo` steps resulting in no modification of a stack `mnhighEnergy` , (it means that conformations are dissimilar to those in the stack and have higher energy). Four actions can be taken:

1. `"heat"`
2. `"stack-jump"` <- default
3. `"random"`
4. `"exit"`

---

interruptAction

action taken upon ICM-interrupt (^\ Control backslash).

1. = `"break loop"`
2. = `"break all loops"` <- default
3. = `"exit macro"`
4. = `"exit to the main macro"`
5. = `"exit all macros"`

---

mfMethod

atom pair selection algorithm used when `"mf"` energy term is calculated by the `show energy`, `montecarlo`, or `minimize` commands.
Allowed values:

- `"intermolecular"` (or `1`) <- default
- `"all"` (or `2`)

(e.g. `mfMethod = 2`)
In contrast to the `"vw"` term, only intermolecular atom pairs are considered by default, since usually intramolecular interactions are calculated with the standard energy terms.
In the `"all"` mode the atom pairs are taken from the van der Waals interaction lists calculated dynamically in the `show energy`, `montecarlo`, or `minimize` commands. All atom pairs except atoms separated by 1 or 2 bonds (so called 1-2 and 1-3 interactions) and within the `vwCutoff` distance are taken into account.
See also: `term "mf", pmf-file`, `mfWeight` .

minimizeMethod

algorithm used for local `energy` minimization which takes place in the `minimize` command, and is a part of one step of a multistep procedure such as `montecarlo`, `ssearch`, and `convert` .
Allowed values:

1. `"conjugate"`
2. `"newton"`
3. `"auto"` <- default

`"conjugate"` means conjugate gradient minimization. Uses analytical first derivatives and takes 6* *n_free_variables* memory.
`"newton"` - quasi-Newton method. It uses analytical first derivatives and takes *n_free_variables*n_free_variables* memory. We recommend this method for energy minimization of small molecules.
`"auto"` <- default; use the more efficient quasi-Newton if the number of free variables (Nof(v_//*) is less than 100 (additional memory requirement of about 2 MB) and switch to the conjugate gradient method if the number of free variables is more than 100.

---

pdbDirStyle

The style of your Protein Data Bank directory/directories. ICM will understand all of the listed styles, including distributions with compressed *.gz , *.bz2 and *.Z files. In all cases, if the `s_pdbDir` variable is set correctly, it is sufficient to refer to the file by its four-character code, e.g.
`read pdb "1abc"`

1. "1abc.pdb"
2. "pdb1abc.ent" "ab/pdb1abc.ent"
3. "ab/pdb1abc.ent.Z"
4. "ab/pdb1abc.ent.gz"
5. "PDB website"

Do not forget to set the right pdb-style in your `_startup` file.

rejectAction

what to do, if the MC procedure rejects `mnreject` trial conformations in a row. Four actions can be taken:

1. " `heat`" <- default choice
2. " `stack jump`"
3. " `random`"
4. " `exit`"

resLabelStyle

style of residue labels invoked by double clicking on the residue or `display residue label` *rs_* command. Possibilities:

1. "`A5`" <- default choice
2. "`Ala 5`"
3. "`ALA 5`"
4. "`Ala`"
5. "`ALA`"
6. "`Alanine 5`"
7. "`5`"
8. "`A`"
9. "` A`"
10. "`Mol`" - displays MOLECULAR name.

See also : `resLabelShift`, `atomLabelStyle` .

ribbonColorStyle

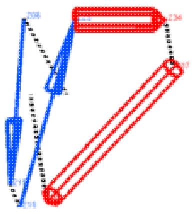
- sets the ribbon coloring scheme.

1 = "type"	default. colors by secondary structure type or explicit color
2 = "NtoC"	colors each chain gradually blue-to-red from N- to C- (or from 5' to 3' for DNA)
3 = "alignment"	if there is an alignment linked to a protein, color gapped backbone regions gray
4 = "reliability"	3D Gaussian averaging with `selectSphereRadius` of alignment strength in space

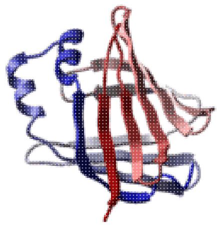
If `ribbonColorStyle` equals to 4, the conserved areas will be colored blue, while the most divergent will be red, and the intermediate conservation areas will be colored white. Example:

```
nice "1eoc.a/"
make sequence a_1.1
read pdb sequence "3pcc.a/"
aa = Align(3pcc_a 1eoc_a)
ribbonColorStyle=3   # color gaps gray
color ribbon
ribbonColorStyle=4   # see alignment strength
color ribbon
```

ribbonStyle



specifies type of representation when `display ribbon` command is used. Options are the following:



1. `"ribbon"` <- default choice
2. `"cylinders"`
3. `"pencils"`
4. `"numbers"`

The first choice is a solid `ribbon` representation.
cylindersThe second representation draws alpha-helices as cylinders. There are two modes depending on the value of the `GRAPHICS.ribbonCylinderRadius` parameter. If `GRAPHICS.ribbonCylinderRadius` is set to zero, the automated radii fitting and helical splitting is engaged. If a helix is too curved, ICM tries to split it into more straight helices. The radius of a helix depends on the helical curvature and is calculated to include all C atoms. Therefore, wide cylinders contain more curved helices.

Alternatively if `GRAPHICS.ribbonCylinderRadius` has a certain non-zero value, this radius will be used.

One can break a helix in any place with the `assign sstructure` command. (e.g. `assign sstructure a_/182 "_"` to break a helix by residue 182).
The third and the fourth, "pencils" and "number" refers to a style where secondary structure elements are represented by vectors (see `Abagyan and Maiorov, 1988`).

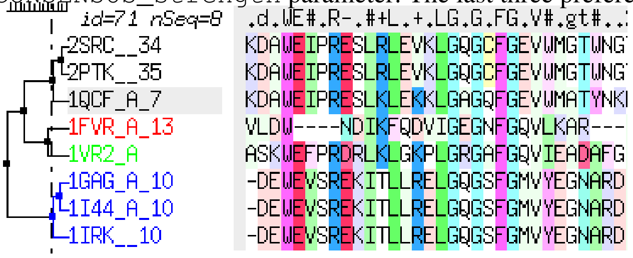
Note The segment parameters must be **pre-calculated** with the `assign sstructure segment` command. The last option ("both") will display both representations of the backbone topology.

sequenceColorScheme

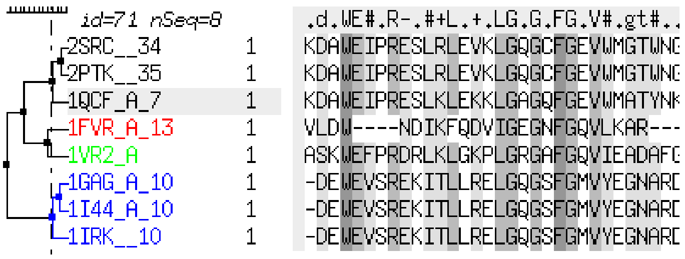
defines the color scheme selection which is used to color alignment in ICM. The following preferences are defined:

1. `"no color"`
2. `"residue type"`
3. `"icm-combo"`
4. `"consensus strength"`
5. `"greyscale"`

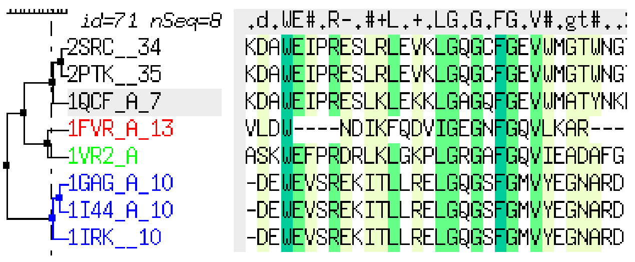
The actual color table containing the correspondence between colors, residues and consensus symbols is stored in the CONSENSUSCOLOR table. The strength of the consensus is regulated by the `CONSENSUS_strength` parameter. The last three preferences are illustrated below.



```
  id=71 nSeq=8   .d.WE#.R-.#+L.+.LG.G.FG.V#.gt#..
 2SRC__34        KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
 2PTK__35        KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
 1QCF_A_7        KDAWEIPRESLKLEKKLGAGQFGEVWMATYNK
 1FVR_A_13       VLDW----NDIKFQDVIGEGNFGQVLKAR---
 1VR2_A          ASKWEFPRDRLKLGKPLGRGAFGQVIEADAFG
 1GAG_A_10       -DEWEVSREKITLLRELGQGSFGMVYEGNARD
 1I44_A_10       -DEWEVSREKITLLRELGQGSFGMVYEGNARD
 1IRK__10        -DEWEVSREKITLLRELGQGSFGMVYEGNARD
```



```
  id=71 nSeq=8        .d.WE#.R-.#+L.+.LG.G.FG.V#.gt#..
 2SRC__34       1     KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
 2PTK__35       1     KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
 1QCF_A_7       1     KDAWEIPRESLKLEKKLGAGQFGEVWMATYNK
 1FVR_A_13      1     VLDW----NDIKFQDVIGEGNFGQVLKAR---
 1VR2_A         1     ASKWEFPRDRLKLGKPLGRGAFGQVIEADAFG
 1GAG_A_10      1     -DEWEVSREKITLLRELGQGSFGMVYEGNARD
 1I44_A_10      1     -DEWEVSREKITLLRELGQGSFGMVYEGNARD
 1IRK__10       1     -DEWEVSREKITLLRELGQGSFGMVYEGNARD
```



```
id=71 nSeq=8   .d.WE#.R-.#+L.+.LG.G.FG.V#.gt#..:
2SRC__34       KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
2PTK__35       KDAWEIPRESLRLEVKLGQGCFGEVWMGTWNG
1QCF_A_7       KDAWEIPRESLKLEKKLGAGQFGEVWMATYNKI
1FVR_A_13      VLDW----NDIKFQDVIGEGNFGQVLKAR---
1VR2_A         ASKWEFPRDRLKLGKPLGRGAFGQVIEADAFG
1GAG_A_10      -DEWEVSREKITLLRELGQGSFGMVYEGNARD
1I44_A_10      -DEWEVSREKITLLRELGQGSFGMVYEGNARD
1IRK__10       -DEWEVSREKITLLRELGQGSFGMVYEGNARD
```

shineStyle

defines how solid surfaces of `cpk` , `skin` and `grobs` reflect light. Possibilities:

1. `"white"` <- default
2. `"color"`

The first option gives a more shiny and greasy look.

surfaceMethod

defines how the `surface energy` is calculated. Options available:

1. `"constant tension"`
2. `"atomic solvation"` <- default choice
3. `"apolar"`
4. `"membrane"`

Explanations:

1. `"constant tension"` means that the energy terms are just the product of the total `solvent-accessible surface` by the `surfaceTension` parameter. This term is intended to represent the surface energy if `electrostatics` takes the solvent polarization energy into account (see `electroMethod`)
2. `"atomic solvation"` option is designed to evaluate the solvation energy purely on the basis of the atomic accessible `surfaces` instead of using the proper `electrostatic` evaluation of the polarization free energy. This fast but approximate scheme was proposed by `Wesson and Eisenberg, (1992)` . Atomic surface parameters derived from the experimental vacuum-water transfer energies are given in the `icm.hdt` file.
3. `"apolar"` option is designed to evaluate the stabilization energy, which is the difference between denatured and folded states. The "atomic solvation" energy should be used with the `van der Waals` term while the "apolar" energy takes it into account and should be used without any other energy terms. The "apolar" atomic surface parameters were derived from the experimental octanol-water transfer energies and are given in the `icm.hdt` file.
4. `"membrane"` option allows one to have a heterogeneous environment with shapes that are 'membrane-like' and shapes with water. The geometrical parameters and shape types are defined by the `TOOLS.membrane` real array. Depending on where an atom is found inside or outside the lipid shape the implicit solvation parameters will be taken from the 7rd column or the 3rd column of the `icm.hdt` file.

Note, that if a part of the system is represented with grid potentials, one needs a special `m_ga` grid map for correct calculations of the surface accessibilities.

The method used to correct the accessibility values by the `m_ga` map can be modified with the `accessMethod` preference. If accessMethod = 3 , the atom accessibilities are multiplied by the `m_ga` value in the vicinity of the atom.

See also:

- `accessMethod`
- `accessMethod`

tzMethod

method of imposing and calculating `tethers`. The three alternatives are the following

1. `"simple"` : equal weight tethers to 3D points
2. `"weighted"` : individual weights are calculated from atomic B-factors by dividing `8*PI`^{2} by the B-factor value. All the weights additionally are multiplied by the `tzWeight` shell variable.
3. `"z_only"` : tethers are imposed only in the Z-direction towards the target Z-coordinate. These type of tethers pulls a molecule into a z-plane. This may be useful if you are trying to generate a flat projection of a three-dimensional molecule.
4. `"function"` : tethers can take a form of distance restraints with individual weights, upper and lower bounds. The three parameters are controlled by the following properties of the **target** atoms (not the source atoms as in the `"weighted"` case): individual weights are directly taked from `bfactor` values, the upper bounds from the `area` fields, and the lower bounds from the `charge` field. To set those values, use the `set bfactor`, `set area` and `set charge` commands respectively. Example:

   ```
   build string "se ala"
   copy a_ tether "tz"
   a_//T    # movable source atoms
   a_//Z    # static destination atoms
   Select(a_ "tz") # also, the destination atoms
   set bfactor a_tz.//* 3. # weight of each tether
   set area a_tz.//* 2. # no penalty within 2A radius around each atom
   set charge a_tz.//* 0. # the lower bound of 0. (can also create repulsion).
   ```

   applying linear force to atoms: to exert a constant force to an atom, set the formal charge of the target atom to a special value of 5. The b-factors will continue to serve as individual force constants and the direction of force will correspond to the vector from the origin to the target pdb-atom with this special value of formal charge.

Example for the `"z_only"` method in which we generate a more or less flat image of a chemical.

```
build smiles "c1c(ccc(c1)N(=O)=O)N2CCC(CC2)=CC(=O)NNC(=O)Nc3cc(ccc3)C(F)(F)F"
tzMethod = "z_only"
set tether a_   # sets tethers to x,y,z=0. coordinates for each atom
minimize "vw,tz" 200
dsChem a_//!h*

#linear force. Use interface to set the linear force flag (formal charge) and bfactors
copy a_ "tzcopy"
tzMethod = "function" # will use bfactor and formal charge features of a_tzcopy. atoms
set tether a_/1/ca a_tzcopy./1/ca  # drag the target atom where you want
set charge formal a_tzcopy./1/ca 5. # number 5. signals ICM to interpret it as linear force
set bfactor a_tzcopy./1/ca 5000.    # the force constant
set tether a_/1/cb a_tzcopy./1/cb   # combine with normal tethers.
display tether a_
minimize v_//?vt* "tz"
```

varLabelStyle

style of labels for free torsions, angles and bonds (i.e. internal variables) `display variable label` *vs* command. Possibilities:

1. `"greek"` <- default choice
2. `"name"`
3. `"value"`
4. `"energy"`
5. `"rings` only"
6. `"value` only"

---

visitsAction

what to do, if one `stack conf`ormation is overvisited, i.e. `mnvisits` has been reached. The following actions are allowed:

1. `"random"`
2. `"heat"` <- default choice
3. `"stackjump"`
4. `"exit"`

Explanation of actions:

- `"heat"` - double the simulation temperature
- `"stackjump"` - jump to the conformation of the least visited slot in the stack.
- `"random"` - randomize all free variables according to the `mcJump` parameter
- `"exit"` - exit the MC procedure

---

vwMethod

specifies the function type of the `van der Waals term ("vw")`. The following three functions can be chosen:

1. `"exact"` <- default choice: $F_{vw} = A/r^{12} - B/r^6$. This is the usual van der Waals formula tending to infinity at r close to 0.
2. `"soft"`: $F_{soft} = F_{vw}$, for $F_{vw} <= 0.$ and $F_{soft} = F_{vw} *(t/(t+F_{vw}))$ for $F_{vw} > 0.$ (repulsion). This form preserves the function for the most populated part of the curve but smoothly reaches the limit `t` (defined by the `vwSoftMaxEnergy` real system variable)
3. "old soft": another smooth approximation with the finite value at r=0, depending on the well depth.

---

webEntrezOption

defines how to interpret the NCBI Entrez links.

1. "none"
2. "g:GenPept" <- default
3. "r:Report"
4. "f:FASTA"
5. "a:ASN.1"
6. "d:Entrez document summary"
7. "m:MEDLINE links"
8. "p:protein neighbors"
9. "n:nucleotide links"
10. "t:structure links"
11. "c:genome links"

See also: `s_webEntrezLink`, `web`, `show html`, `write html`.

---

wireStyle

style of the `display wire` mode. The choices are the following:

1. "wire" <- default choice
2. "chemistry"
3. "tree"

Style "chemistry" shows different types of chemical bonds. Style "tree" shows a directed graph of the ICM-molecular tree. Yellow triangle indicates the entry atom of an ICM object. The tree can be rerooted with the `write library a_newEntryAtom` command. The topology of the complete tree including the virtual atoms can be shown with the `display virtual` command.
Note: The "tree" graph does **not** exist for objects of non-ICM type, e.g. those created by the `read pdb` command, and this preference will have no effect. The tree representation elucidates the ICM topology graph imposed on molecules and is crucial in the `modify` command, since it removes a branch up-tree from the specified entry atom, and replaces it by another branch. Use `Ctrl-W` to toggle between these styles (see `set key` command). The line width is controlled by the `lineWidth` parameter.

---

xrMethod

The penalty function of correspondence between observed and calculated structure factors.

1. "corr Fc:Fo" <- default
2. "corr Fc2:Fo2"

Tables

The following predefined icm-shell tables are collections of different `icm shell objects` related to a certain topic. Note that these tables (as opposed to user-defined ICM `tables`) usually only have the header section. You can `show` and `list` them. You can also change any table element by the usual icm assignment:
Examples:

```
 IMAGE.color = yes       # this member is a logical
 IMAGE.stereoBase = 2.5  # redefine real distance between stereo panels
```

CONSENSUS

The consensus *symbol* is established if the percentage of specified *residues* in a give column exceeds the *fraction* given in the 2nd column. In rows were we provide two symbols (e.g. "-n"), the first (e.g. '-') is used in alignment representations, while the letter form of this symbol (e.g. 'n') is used in residue selections, (e.g. `a_/Cn`) The first matched consensus condition takes precedence. In an example below, if `Q` is found in more than 85% or sequences, its consensus symbol is `Q`, if the percentage is between 60 and 85, the symbol becomes `q`, and if no consensus is establish, the symbol becomes the dot character ('.').

```
#>T CONSENSUS
#>-symbol------fraction----residues---
   A           85          A
   C           85          C
   Q           85          Q
...
   d           60          ND
   -n          70          ED
   +o          60          RK
   gj          60          G
   q           70          Q
   p           60          P
   t           60          TSN
   "#h"        85          WLVIMAFCYHP
   %f          65          WLVIMAFCYHP
   " g"        85          -
```

See also: `CONSENSUSCOLOR` , `CONSENSUS_strength` , `color alignment` *rs_* , `ribbonColorStyle` .

CONSENSUSCOLOR

contains coloring schemes of residues according a multiple sequence alignment (see the `align` command). This table is saved together with the GUI preferences. Residue color is defined by two factors: its type, as listed in the first column of the table, AND the consensus character under which this residue is aligned. The consensus symbols are defined by the `CONSENSUS` table and are listed in the third column of the CONSENSUSCOLOR table that is loaded from the $ICMHOME/CONSENSUSCOLOR.tab file. In this scheme the same residue can be colored differently depending on the alignment in a current position.

```
#>T CONSENSUSCOLOR
#>-residue-----color-------symbols----
   *           *           *          # separator between sections
   ED          "#ff0000"   "-EDp"     # E and D residues will be colored red under '-','E' or '
   KR          "#0000ff"   "+KRp"
```

See also: `CONSENSUSCOLOR` , `CONSENSUS_strength` , `color alignment` *rs_* , `ribbonColorStyle` , `set color`.

FILTER

contains filters which can be applied to the input stream in the `read` command. Components have names corresponding to standard file name extensions; their `string` value is a unix filter. Token **%s** is a placeholder for the file name. The provided defaults can be redefined in your `_startup` file. You can also add your own extensions and filters by doing the following:

```
 z = "pcat %s"     # define the action for the unix packed files
 group table append FILTER header z  # append new filter to the structure
```

The mechanism ICM employs allows one to keep the transformed files intact and avoid creating temporary files when possible (e.g. uuencode unix command always creates an output file). Existing extensions and defaults are given below. You may need to redefine the defaults by adding the exact path to the utility or using alternatives.

FILTER.Z

allows you to read the compressed files (*.Z) directly leaving the compressed file intact. The default value: `"zcat %s"` . If you do not have zcat utility, try
FILTER.Z = "uncompress -c %s"

FILTER.gz

The default value is `"gunzip -c %s"` .

FILTER.uue

The default value is

```
"sed 's:begin .*:begin 600 /tmp/UUPtm:' %s | uudecode && cat /tmp/UUPtm && rm -f /tmp/UUPtm"
```

This works for UNIX file system, write your own on the PC, if needed.

FTP

`table` which controls reading from `ftp`.

FTP.createFile

(default `no`). This flag is not active yet. The file is always created in the `s_tempDir` directory.

FTP.keepFile

(default `no`). If `yes`, the temporary file is kept in the `s_tempDir` directory. Otherwise the file is deleted.

FTP.proxy

`string` name of the proxy server for ftp connections through firewall. Default: "" (empty string).

String format: ftp.proxy.host.com[:port]

HTTP.proxy

`string` name of the proxy server for http connections through firewall. Default: "" (empty string).

String format: [user[:pass]@]http.proxy.host.com[:port]

HTTP.ignoreProxyDomains

`string` with ';' separated list of domains where `HTTP.proxy` should not be used.

Example:

```
HTTP.ignoreProxyDomains = "localhost;*.molsoft.com" # local host + everything which ends with
```

GRAPHICS

display parameters for different graphics representations.

GRAPHICS.atomLabelShift

a non-negative integer number of spaces preceding an atom label. This parameter is useful for displaying labels next to a solid representation, such as `xstick` or `cpk`.

See also: `GRAPHICS.resLabelShift` Default (0)

GRAPHICS.atomValueCircles

GRAPHICS.atomValueCircles allows one to display a circle with a **positive** value on every atom for those atom fields:

1. = `"none"`
2. = `"field"`
3. = `"b"`
4. = `"occupancy"`
5. = `"area"`

The radius of the circle and the value tranformation upon changing hydrogen display is defined as follows:

- `"field"` shown as is, the values from undisplayed hydrogens are accumulated
- `"b"` radius is the b-factor value divided by a 100.
- `"occupancy"` shown as is
- `"area"` radius is the b-factor value divided by a 40., the values from undisplayed hydrogens are accumulated

The Escape button resets the preference to `"none"`.

Example:

```
show surface area a_
set area a_//n* 100.  # just to show that it can be custom set as well
GRAPHICS.atomValueCircles = "area"
display new
```

GRAPHICS.ballRadius

radius (in Angstroms) of a small ball `display`ed as a part of `ball` or `xstick` graphical representations of a molecule.
Default (0.15)

GRAPHICS.ballStickRatio

A default ratio of ball and stick radii. This ratio is applied when the styles are switched from the GUI xstick toolbar.
Default (1.4)

GRAPHICS.clashWidth

relative width of a displayed `clash` . This parameter can be changed from the **File/Preferences/DisplayGeneral** menu.
See also: `lineWidth` , `GRAPHICS.grobLineWidth` , `GRAPHICS.hbondWidth` , `GRAPHICS.mapLineWidth` , `lineWidth` .
Default (1.)

GRAPHICS.chainBreakStyle

controls how missing residues in a missing protein fragment are displayed (in `ribbon` style). Now the gaps can be ignored or shown as ribbon bullets. Thus, available choices are the following:

1. = "none" # nothing is displayed
2. = "bullets" # gap is show as bullets, the number of bullets depends on the number of missing residues

The gaps are labeled according to the `GRAPHICS.chainBreakLabelDisplay` parameter.

Individual treatment of the chain gap display. The ribbon chain break display can also be suppressed at the molecular level with the `set property "nobreaks"` command, e.g.

```
set property "nobreaks" a_1
```

See also:

- `GRAPHICS.chainBreakLabelDisplay`
- `ribbon`

GRAPHICS.chainBreakLabelDisplay

controls how the number of missing residues in a missing protein fragment is displayed (usually as a `ribbon`). ICM tries to draw one bullet for each missing residue. In the `auto` mode the label is displayed only if the number of bullets is different from the number of missing residues.

1. = "none" # the label is not shown
2. = "all" # the label is always shown.
3. = "auto" # shows label if N bullets != N missing residues

See also: `GRAPHICS.chainBreakStyle`

GRAPHICS.cpkClipCaps

preferences to control the way the `cpk` spheres are being displayed when cut by a clipping plane.

1. = "none"
2. = "stencil" = "explicit"

Both capping methods have some side effects and slow down the graphics performance. User discretion is advised. See also:

GRAPHICS.displayLineLabels

enables/disables the display of edge lengths (inter-point distances) of a `grob` generated with the `Grob( "distance"` ..) function. This parameter can be changed from the **File/Preferences/DisplayGeneral** menu.
See also: `Grob` ("distance" ..)
Default (yes)

GRAPHICS.displayMapBox

controls if the bounding box of a map is displayed (see `display map`).
Default (yes)

GRAPHICS.dnaBallRadius

DNA bases in ribbon representation are shown as balls controlled by the above `real` parameter. You can undisplay them with the: `undisplay ribbon base` command.
Default: 1.5

GRAPHICS.dnaRibbonRatio

`real` ratio of depth to width for the DNA `ribbon` .
Default: 0.3

GRAPHICS.dnaRibbonStyle

`GRAPHICS.dnaRibbonStyle` = "`complex shapes`" a method of schematic/simplified representation of DNA or RNA in which the bases are shown as:

1. = "ball"
2. = "flat shapes"
3. = "complex shapes"

GRAPHICS.dnaRibbonWidth

`real` width (in Angstroms) of the DNA `ribbon` .
Default: 2.

GRAPHICS.dnaRibbonWorm

`logical` which, if yes, makes the DNA backbone `ribbon` round, rather than rectangular.
Default: no

GRAPHICS.dnaStickRadius

`real` radius of the sticks representing bases in DNA `ribbon` .
Default: 0.72

GRAPHICS.formalChargeDislplay

a preference regulating the formal charge visualization of the visible atoms:

1. = "none" : do not display formal charges
2. = "all" : label all formally charge atoms
3. = "integer only" : skip fractional formal charges
4. = "ligand only" : do not display charges on polymers, display them only on 'hetatm' compounds

See also:

- `GRAPHICS.occupancyDisplay`

GRAPHICS.grobDotSize

default radius of a dot/vertex in a grob.

See also: `lineWidth` , `GRAPHICS.grobLineWidth`

Default: `3.0`.

GRAPHICS.grobLineWidth

relative width of displayed lines of 3D meshes (`grobs`). Also affects the interatomic distance display. This parameter can be changed from the **File/Preferences/DisplayGeneral** menu.
See also: `lineWidth` , `GRAPHICS.clashWidth` , `GRAPHICS.hbondWidth` , `GRAPHICS.mapLineWidth` .
Default (1.)

GRAPHICS.hbondStyle

determines the style in which hydrogen bonds are displayed. Here hbond-Donor, Hydrogen, and hbond-Acceptor atoms will be referred to as D, H and A, respectively,

```
GRAPHICS.hbondStyle = "dash"
       1 = "dash"      # the default choice. Just a line
       2 = "length"    # show the D-A distance in addition
       3 = "length and angle"  # show both the distance and the 180. - <D-H.. A> angle
```

The best possible value for a non-linearity angle is 0. . The display dialog has a small button to roll through this preference. See also: `GRAPHICS.hbondWidth` .

GRAPHICS.hbondRebuild

This preference determines if the Graphics user interface (GUI) hbond button will display hydrogen bonds dynamically (the bonds will change as the coordinates change), or a `parray` of pairwise distances will be generated separately. It also regulates if intramolecular bonds are suppressed.

1. "static"
2. "dynamic"
3. "intermolecular"

Intermolecular bonds imply the `static` method.

See also:

- `make hbond`
- `display hbond`
- `Table(distobj distance)`

GRAPHICS.hbondMinStrength

`GRAPHICS.hbondMinStrength` parameter determines the hbond strength threshold for hbond display. The strength value is between `0.` and `2.` By changing 1. to 0.2 you will see more weak hydrogen bonds.

This parameter can be changed from the GUI hbond button popup-menu.

See also: `GRAPHICS.hbondAngleSharpness` Default: 1.

GRAPHICS.hbondAngleSharpness

GRAPHICS.hbondAngleSharpness determines how the strength depends on the D-H...A(lone pair) angle. The preference can be found the general Preferences menu

Default: 1.7

GRAPHICS.hbondBallPeriod

This parameter defines the distance between centers of spheres of hbonds divided by the diameter of the sphere.

GRAPHICS.hbondBallStyle

GRAPHICS.hbondBallStyle parameter controls the size of the hbond spheres and the gradient of those sizes. The master size is a fraction of the GRAPHICS.ballRadius .

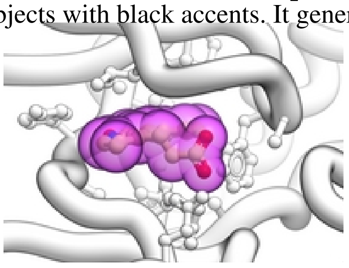
The following possibilities are implemented:

1. = "even" : all hbond-spheres have the same size
2. = "telescopic" : a zoom from donor to acceptor
3. = "by energy" : better hbonds are shown with thicker spheres
4. = "by atom size" : the radii form a gradient between ball radii of the hbonded atoms.

GRAPHICS.hbondWidth

relative width of a displayed hbond . This parameter can be changed from the **File/Preferences/DisplayGeneral** menu.
See also: lineWidth , hbond display hbond, GRAPHICS.grobLineWidth , GRAPHICS.clashWidth , GRAPHICS.mapLineWidth .
Default (1.)

GRAPHICS.sketchAccents

logical GRAPHICS.sketchAccents if yes , activates a drawing mode that highlights the boundaries of objects with black accents. It generates a Edouard Manet type visual effect (e.g. Olimpia, 1863).  Example in which the current object is shown as ribbon with ligands in cpk and surrounding side chains as xsticks:

```
  GRAPHICS.sketchAccents = yes
  GRAPHICS.ribbonWorm = yes
  GRAPHICS.wormRadius = 1.
  GRAPHICS.quality = 20
  GRAPHICS.stickRadius = 0.25
  GRAPHICS.ballRatio = 1.1
  GRAPHICS.transparency[1] = 0.6
  GRAPHICS.hetatmZoom = 1.1
  color background white
  color residue label black  # if you decide to display them
  display ribbon white
  display a_H cpk magenta
  display a_H xstick
  display Res(Sphere( a_H a_!H,W//!c,ca,n,o,h*  )) & !a_*.//n,o,h* xstick white
  color xstick a_!H,W//n* lightblue
  color xstick a_!H,W//o* lightpink
#  display residue label a_//DX
```

GRAPHICS.hetatmZoom

The default `ball` and `stick` radii of a ligand can be different by the `GRAPHICS.hetatmZoom` factor. This makes a better ligand view since the ligand stands out from the surrounding protein atoms.
See also, `icm.clr` file about changing the default color for carbon atoms in ligands (a.k.a. `hetatm`) (`atom H` *color*) Default (1.5)

GRAPHICS.hydrogenDisplay

determines the default hydrogen display mode for the `display` command.

```
 GRAPHICS.hydrogenDisplay = "polar"
        1 = "all"   # all hydrogens are shown
        2 = "polar" <-- current choice  # polar displayed, the non-polar hidden
        3 = "none"  # no hydrogens are displayed
```

GRAPHICS.light

a `rarray` of 13 elements between 0. and 1. which controls the main properties of lighting model in GL. The sections of this array can be changed with four sliders of the Display tab in a top tool bar. The following elements are defined:

Elements	Property	Range	Default	Comment
GRAPHICS.light[1]	shininess	0.,1.	1.	property of the solid material
GRAPHICS.light[2:4]	ambient light	0.,1.	{0.15 0.15 0.15}	intensity of RGB for ambient light
GRAPHICS.light[5:7]	diffuse light	0.,1.	{0.6 0.6 0.6}	intensity of RGB for diffuse light
GRAPHICS.light[8:10]	specular light	0.,1.	{0.35 0.35 0.35}	intensity of RGB for specular light
GRAPHICS.light[11:13]	emission	0.,1.	{0. 0. 0.}	intensity of RGB for emitted light

The first element defines the shininess of solid surfaces such as `cpk`, `ribbon`, `ball`, `stick`, `xstick`, and `skin` when they are `display`ed. The other elements contain triplets of R,G,B (red green blue) values from 0. to 1. for the four types of visual effects. If R,G and B channels do not have equal intensity (e.g. `GRAPHICS.light[5:7] = {0.2 0.2 0.6}`) the corresponding light effect will have color (blue in the example above).
To re-render the solid graphics with new parameters, use the

```
display new reflection
```

command.
Example:

```
build string "se his trp glu"
display cpk
color background blue
GRAPHICS.light[5:7] = {0.2 0.2 0.6}
display new reflection
```

See also: `GRAPHICS.lightPosition` .

GRAPHICS.lightPosition

X,Y and Z posiion of the light source in the graphics window. The X and Y coordinates are usually slightly beyond the [-1. 1] range where [-1.,1.] is the size of the window, and the Z position is perpendicular to the screen and is set to 2. (do not make it negative). The default values are the following:

```
GRAPHICS.lightPosition = {-1.,-2.,2.}
```

See also: `GRAPHICS.light` .

GRAPHICS.mapLineWidth

relative width of lines and dots of a displayed `map` . This parameter can be changed from the **File/Preferences/DisplayGeneral** menu.
See also: `lineWidth` , `GRAPHICS.grobLineWidth` , `map` , `GRAPHICS.hbondWidth` .
Default (1.)

GRAPHICS.occupancyDisplay

`preference` controlling if and how the partial or zero atom occupancies are displayed. The abnormal occupancies are shown as circles around atoms. These following values are allowed.

1. = "none" # nothing is displayed
2. = "circle" # a circle is displayed
3. = "label" # a circle and a label with the value (zero values are not shown)

A silly example.

```
GRAPHICS.occupancyDisplay="label"
build string "se ala his"
set occupancy a_//n* 0.5
display xstick
```

See also:

- ♦ `GRAPHICS.occupancyRadiusRatio`
- ♦ `GRAPHICS.formalChargeDisplay`

GRAPHICS.occupancyRadiusRatio

The radio of a circle showing non-1. occupancy atoms to the van der Waals radius of the atom.

Default value: 1.5

See also:

- ♦ `GRAPHICS.occupancyDisplay`

GRAPHICS.quality

`integer` parameter controlling quality (density of graphical elements) of such representations as cpk, ball, stick, `ribbon` . Do not make it larger than about 20 or smaller than 1. This parameter supersedes the previous `ballQuality` parameter.
We recommend to make this parameter at least `15` if you want to make a high quality image. You can also increase the number of image resolution by making the image window 2,3,4 times larger (in the example below it is 2 times larger) than the displayed window.

```
GRAPHICS.quality = 15
display ribbon

# press Ctrl-D  for the fog effect, move clipping planes, change fogStart

write image png window=2*View(window)
```

Default: 5.

GRAPHICS.rainbowBarStyle

determines if and where the color bar will appear after a molecule is colored by an array. Coloring by an array is one of the options of the `display` and `color` commands.

1. = "left" <- default choice
2. = "right"
3. = "no text"
4. = "no bar"

The bar can be dragged (use middlebutton), removed (point into the bar and press `BACKSPACE`), just like a string label. To assign your own numbers to the bar, you may choose option `"no` text" and use several display s_label commands. The bar, if displayed, is exported to TIF, RGB images and postscript.

GRAPHICS.resLabelDrag

if yes, enables dragging of the displayed residue labels with the middle mouse button. The labels can be reset to their initial positions with the set residue label distance *rs_* command. The initial position is defined by the relative displacements of {0. 0. 0.} from the special "residue label-carrying" atom of the residue, see the set label as_ command. See also resLabelStyle
Default (no).

GRAPHICS.resLabelShift

a non-negative integer number of spaces preceding a residue label. This parameter is useful for displaying residue labels next to a solid representation, such as xstick , ribbon or cpk.

See also: GRAPHICS.atomLabelShift, GRAPHICS.resLabelDrag and s_labelHeader

Default (no).

GRAPHICS.ribbonCylinderRadius

GRAPHICS.ribbonCylinderRadius is the real radius of helical cylinders in schematic protein topology display for the ribbonStyle = "cylinders" preference.

The radius can be set to zero for an automated, variable-radius mode or to a specific radius. Example:

```
read pdb "1crn"
ribbonStyle = "cylinders"
GRAPHICS.ribbonCylinderRadius = 2.2
display ribbon
```

See also:

- ribbon
- ribbonStyle

GRAPHICS.ribbonGapDistance

The minimal distance between the first atoms of the neighboring residues when the ribbon gap will be drawn if l_breakRibbon logical is set to yes.

Default: 4.

GRAPHICS.ribbonRatio

real ratio of depth to half-width for the protein ribbon .

Warning: note that this parameter influences GRAPHICS.wormRadius if GRAPHICS.ribbonWorm is set no no . In this case GRAPHICS.wormRadius will be redefined as GRAPHICS.ribbonWidth * GRAPHICS.ribbonRatio automatically.
Default: 0.3

GRAPHICS.ribbonWidth

`real` width of the protein `ribbon` .
Default: 1.

GRAPHICS.ribbonWorm

`logical` parameter, if yes, makes the ribbon round, rather than rectangular.

Default: no

GRAPHICS.rocking

`preference` with the following options:
1. = "X-rocking"
2. = "Y-rocking"
3. = "Xy-rocking"
4. = "xY-rocking" = "X-rotation"
5. = "Y-rotation"

See also, `GRAPHICS.rockingRange` and `GRAPHICS.rockingSpeed` , `display rotate`
Default: 4

GRAPHICS.rockingRange

`real` value of rocking range.

Default: 1.

GRAPHICS.rockingSpeed

`real` value of rocking or rotation speed.

Default: 1.

GRAPHICS.selectionLevel

preference for the selection level of `as_graph` selection. The atoms, residues, molecules or objects selected interactively in the graphics window are automatically stored in the `as_graph` variable. The preference may have the following values.

```
GRAPHICS.selectionLevel = "atom"
      1 = "object"
      2 = "molecule"
      3 = "residue"
      4 = "atom" # default
      5 = "variable"
```

The `GRAPHICS.selectionLevel` can be switched either interactively, e.g.

```
GRAPHICS.selectionLevel = 3
```

or from GUI by selecting the level combo box with the following choices: `O` (object), `M` (molecule), `R` (residue), `x` (atom), or an icon of a torsion (variable).

GRAPHICS.selectionStyle

preference for the style in which the graphical selection is shown. The preference may have the following values.

```
GRAPHICS.selectionStyle = "color"
```

```
1 = "none"
2 = "cross" # the default choice
3 = "color"
4 = "both"
```

In the 1-st mode (`"none")` only a single selection mark is shown. It is convenient when you do not want multiple selection marks to overwhelm the image. The 3-rd mode is incovenient if you want to try different colored displays for the selected fragments.

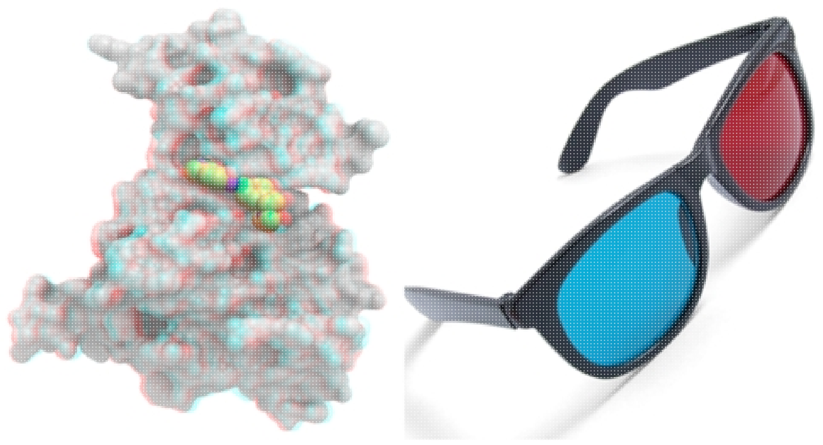
GRAPHICS.stereoMode

1. "up-and-down"
2. "line interleaved" # current choice
3. "in-a-window"
4. "Sharp"
5. "Anaglyph"

a simple hardware stereo mode for workstations with a horizontal frame splitter. In the `"up-and-down"` mode a longer frame with two stereo images on top of each other is generated and the two halves are then superimposed with the splitter. This mode does not require anything from a graphics card, but does require a frame splitter. A frame splitter box was connected between a monitor and a graphics card output. This mode has an unpleasant side effect, the rest of the screen (beyond the OpenGl window) becomes stretched and the lower part of the screen is superimposed on the top half.
The `"line` interleaved" mode can be used with a new type of frame splitter at the line level. In this case the odd lines from one stereo-image are interleaved with the even lines of another. The side-effect of this mode is that the intensity is reduced in half since at each moment one sees only one half of the lines. The splitter device for this mode can be purchased from Virex (www.virex.com). This mode produces a dark stereo image but is easily available (requires stereo goggles, e.g. from Virex).
The "in-a-window" mode is used in SGI workstations and in a Linux workstation with an advanced graphics card supporting a quad graphics buffer. In this mode the hardware stereo regime applies only to an OpenGl window. This is the best mode but it requires an expensive graphics card (plus the stereo goggles).
Note: LCD screens can not display a stereo image since the image is not continuously updated at high frequency. This technical problem may be solved in the future (so we hear).

The "Sharp" option is for SHARP manufactured 3D screens.

The "Anaglyph" option is for the stereoscopic 3D effect achieved by means of encoding each eye's image using filters of different (usually chromatically opposite) colors, typically red and cyan. The Anaglyph option is the easiest to used with inexpensive 3D glasses and and without any expensive 3D compatible hardware or monitors.



GRAPHICS.stickRadius

radius (in Angstroms) of a cylinder `display`ed as a part of `stick` or `xstick` graphical representation of a molecule.
Individual (residue-wide) control of stick radii.
In order to modify the default values of the radii from the command line use the `set xstick` *r_newradius* command For example:

```
set xstick a_/13:15 0.5
```

In this case the ball radius will be changed according to the radio of the **default** parameters (e.g. `GRAPHICS.ballRaduis/GRAPHICS.stickRadius`)
Default (0.4).

GRAPHICS.surfaceDotSize

Determines the size of the dot on the solvent accessible graphical `surface area` . This surface is controlled by the `GRAPHICS.surfaceProbeRadius` radius. If this parameter is changed, use

```
display surface refresh
```

to update the dot size.
Default (2.0).

GRAPHICS.surfaceDotDensity

Determines the number of dots per square Angstrom on the graphical solvent accessible `surface area` . Do not confuse this parameter with `surfaceAccuracy` . The latter controls the surface and energy calculation and does not affect the displayed surface.

See also: `GRAPHICS.surfaceProbeRadius` and `GRAPHICS.surfaceDotSize`. Default (10).

GRAPHICS.surfaceProbeRadius

An increment to the van der Waals radii of atoms at thich the dotted atomic surface is calculated. It is used by the `display surface` command to display dotted van der Waals surface. If the `GRAPHICS.surfaceProbeRadius` is set to 1.4 the surface becomes equivalent to the solvent accessible surface with a probe of 1.4A (e.g. in `show surface area`) .

Note, that in contrast to `GRAPHICS.surfaceProbeRadius`, the `vwExpand` parameter is used for *calculations* of the solved accessible areas (e.g. `show surface area`).

Default (0.1)

GRAPHICS.transparency

Two parameters regulating the transparency of grobs.

- ♦ `GRAPHICS.transparency[1]` contains a value from 0. to 1. representing the level of transparency (0. solid, 0.99 - invisible)
- ♦ `GRAPHICS.transparency[2]` contains the brightness of the transparent surface from 0. to 1.

Grobs can also have individual transparency values (see `set grob` and `make grob skin`)

GRAPHICS.wormRadius

radius of `coil`ed segments (i.e. those where the secondary structure is marked as "_") of a polypeptide chain in `ribbon` representation.

Warning: this parameter behaves as independent only in the `GRAPHICS.ribbonWorm` is `yes` . Otherwise, in case of a mixed ribbon representation display, ICM will reset the radius to the product of `GRAPHICS.ribbonWidth` and `GRAPHICS.ribbonRatio` in order to match thickness of the ribbon and the connecting coil.
Default (0.3).

GRID

parameters for the grid energy calculations (see also `"gh,gc,ge,gs,sf" energy terms`).

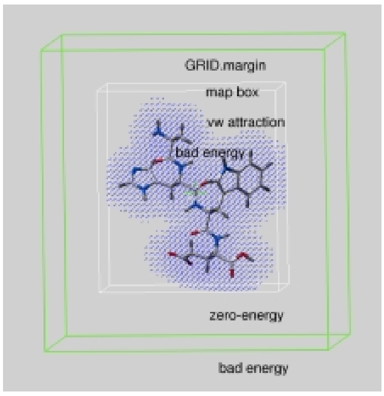
GRID.gcghExteriorPenalty

A preference to allow automatically impose a repulsive penalty outside the area covered by the van der Waals maps (`m_gc` and `m_gh`).

1. = `"repulsive"` <- the default
2. = `"zero"`

In the default mode a volume penalty is imposed automatically outside the map box expanded by the `GRID.margin` . The penalty potential is set to the `GRID.maxVw` value.

GRID.margin



`real` parameter determining the extra penalty-free space around the `map` bounding box if `GRID.gcghExteriorPenalty = "repulsive"` (see above). For any atom which gets outside the map-bounding box expanded by `GRID.margin`, its grid van der Waals energy (`"gc"` or `"gh"`) is penalized by the `GRID.maxVw` value. This is the same penalty value which atoms get if they severely clash with other atoms.

Therefore, if you set up grid energy calculations it is essential either to create a big enough box or set a sufficient margin to allow ligand rearrangements near the receptor surface. If `GRID.margin` is very large, your ligand will be "on the loose" and may spend too much time flying in open air. It is recommended that the margin is not larger than the diameter of your ligand. Default: `0.00` A

See also: `GRID.gcghExteriorPenalty`

GRID.maxEl

`real` truncation parameter. Default: `20.0` kcal/mole.

GRID.minEl

Default: -20.0 kcal/mole.

GRID.maxVw

The truncation level of the van der Waals repulsion energy pre-calculated in the `"gc"` grid energy term. This number also is used as a penalty for the atoms outside the map box expanded by `GRID.margin` .
Default: 3.0 kcal/mole.

GRID.gpGaussianRadius

The radius of the density Gaussian used to generate the property maps. Default: 1.2 A. See also:

- `term "gp"`

- ♦ `set type property` *I_atomTypes R_upToSevenWeights*
- ♦ `make map potential` "gp"

GROB

Parameters related to `graphics objects`. See also the `Grob` family of functions.

GROB.atomSphereRadius

default radius (in Angstroms) which is used to select a patch on the surface of a grob. Used in the `color` *grob as_selection color* command. See also: `Grob( g R_6)` function to return a patch of certain color.
Default: `4.0.`

GROB.relArrowSize

a `real` relative arrow size ([0.,1.]). Default: `0.2.`

GROB.arrowRadius

a `real` arrow radius in Angstroms used by the `Grob`("ARROW", R_) function.

See also: `makeAxisArrow` , `GROB.relArrowHead` and `GROB.`\`relArrowSize . Default: `0.5.`

GROB.relArrowHead

a `real` ratio of the arrow head radius to the arrow radius. This parameter is used by the `Grob`("ARROW", R_) function. Default: `3.0.`

GROB.contourSigmaIncrement

a `real` increment in the sigma level used to re-contour an electron density map using the `make grob` *`m_eds`* `add` *`r_increment`* command.

This parameter is used in the GUI when plus and minus are pressed.

Default (0.1)

GUI

This parameter table contains some settings for the GUI (see below). Most of the settings are stored automatically in the `s_userDir` + `"/config/icm.cfg"` file

To read about generating dialogs and menus from ICM see `gui` .

GUI.autoSave

a `logical` to activate the saving of the content of the session every `GUI.autoSaveInterval` seconds. If the program exists unexpectedly or crashes the session then can be restored.

GUI.autoSaveInterval

The number of seconds the session will save itself into a backup file as a precautionary measure against an unexpected termination or crash (see `GUI.autoSave`) .

GUI.defaultLayoutAction

The behavior of ICM windows upon clicking the "Default Layout" button (third at the left bottom corner). The "Main" location is the upper right window that gets maximized upon pressing the fourth button.

```
GUI.defaultLayoutAction = "3D to Main"
      1 = "3D to Main" <-- current choice
      2 = "Tables to Main"
      3 = "Alignmnets to Main"
      4 = "Html To Main"
      5 = "Keep Main"
```

GUI.tableRowMarkColors

E.g.

```
GUI.tableRowMarkColors = "#ff4444/#ffbb44/#44ff44/#4499ff/#ff44ff"
```

GUI.windowLayout

defines one of the two alternative ICM-panel layout styles:

1. = `"traditional"`
2. = `"widescreen"`

The widescreen layout will extend the graphics window vertically by shortening the terminal window to the width of the workspace panel.

GUI.workspaceStyle

defines the look for objects and molecules in the ICM workspace panel.

1. = "all"
2. = "simple"

GUI.workspaceTabStyle

allows one to change the style of ICM-object tabs created in the workspace panel of ICM GUI.

1. = "icon title" # default
2. = "icon"
3. = "title"

GUI.workspaceFolderStyle

defines how many hierarchical levels of 3D molecular objects are shown in the Workspace panel.

1. = "closed"
2. = "object"
3. = "molecule" = "residue"

IMAGE

table contains settings used by the following commands creating image files:

- `write image`,
- `write postscript`,
- `display trajectory image`.

IMAGE.quality

this `integer` parameter allows one to improve quality of vectorized postscript images saved by the `write postscript` command. Actually this parameter only changes one number in the header of a postscript file. You can also manually edit the file to correct this number. This number

defines the number of divisions of larger triangles into smaller ones accompanied by interpolation of colors which occurs during printer interpretation of the postscript stream to provide smooth continuous transitions. The optimal value of this parameter depends on the maximal triangle size. It may grow as large as 100 for a single triangle on a page. Typically for a molecular image with molecular surface `IMAGE.quality=3` is sufficient.
Important. Do not set the parameter to values higher than 5 for the molecular image, your printer will die!
Default: `3`

IMAGE.printerDPI

this `integer` parameter the printer resolution in Dot Per Inch (DPI). Important for the `write image postscript` command.
Default: `300`

IMAGE.lineWidth

this `real` parameter specifies the default line width for the postscript lines.
Default: `1.0`

IMAGE.scale

`real` variable. If non zero, controls the image scale with respect to the screen image size. The screen image resolution (or Dots-Per-Inch) is usually 72. Let's assume printer DPI to be 300 (see the `IMAGE.printerDPI` parameter). In this case `IMAGE.scale=1.` will make the printed image the same pixel size (which is about 4 times smaller) than the screen image. For pixel images saved by `write image postscript` command integer `IMAGE.scale` values (2., 3., 4.) are preferable. That is what auto mode (IMAGE.scale=0.0) is trying to do. This consideration is NOT important for the vectorized postscript images created by the `write postscript` command.
Default: `0.0` (i.e. auto mode: maximum size fitting the page in given `IMAGE.orientation`)

IMAGE.stereoBase

`real` variable to define the stereo base (separation between two stereo panels) in the `write image postscript` and `write postscript` command.
Default: `2.35` inches, (~ 60mm)

IMAGE.stereoAngle

`real` variable to define stereo angle (relative rotation of two stereo images) in the `write image postscript` and `write postscript` command.
Default: `6.0` degrees.

IMAGE.gammaCorrection

`real` variable to to lighten or darken the image by changing the *gamma* parameter. A gamma value that is greater than `1.0` will lighten the printed picture, while a gamma value that is less that `1.0` will darken it. You may adjust your gamma correction parameter for your printer with respect to your display and add this setting to the `_startup`
Default: `2.0`

IMAGE.color

`logical` to save color or black_and_white ('bw') images. You can override this parameter by using the explicit `bw` option in the `write image` command.
Default: `yes`

IMAGE.compress

`logical` to toggle simple lossless compression, standard for .tif files. This compression is required to be implemented in all TIFF-reading programs.
Default: `yes`

IMAGE.generateAlpha

`logical` to toggle generation of the alpha (opacity) channel for the SGI `rgb`, `tif` and `png` image files to make the pixels of the background color transparent.
Be careful. The alpha channel is set to 1. for every pixel in your image which has the same color as the background. Therefore there is a danger that the same color will be accidentally used inside your image. If you nevertheless want to generate the alpha-channel, use a rare color your background (not black, but rather green, e.g. `rgb = {0.,0.976,0.}.`
Default: `yes`

IMAGE.stereoText

`logical` to make text labels for only one panel or both panels of the stereo diagram.
Default: `yes`

IMAGE.previewer

a `string` parameter to specify the external filter which creates a rough binary (pixmap) postscript preview and adds it to the header of the ICM-generated high resolution bitmap or vectorized postscript files saved by the `write image postscript`, and `write postscript` , respectively . This preview information is compliant with EPSI (encapsulated Postscript interchange file format) and is useful to see a draft image instead of a empty rectangle upon inclusion of the postscript file into other drawing and imaging software like IRIS showcase.
Default: `"gs -sDEVICE=pgmraw -q -dNOPAUSE -sOutputFile=- -r%d -&#8211 %s"`

IMAGE.previewResolution

`integer` resolution of the rough bitmap preview added to the vectorized postscript file in *lines per inch*. Recommendations:

- ♦ 10 - very rough (1/10th of an inch)
- ♦ 20 - a reasonable preview but no fine details
- ♦ 30 - a fine preview, do not increase it any higher since the file will become too large.

IMAGE.lineWidth2D

`integer` thickness of bonds in chemical 2D drawing upon the Copy Image command. This is useful for cutting and pasting from ICM to external documents. Default: 1.5 pixels

IMAGE.bondLength2D

`real` length of a chemical bond (in inches) in chemical 2D drawings upon the Copy Image command. To make your molecule large, increase it. This is useful for cutting and pasting from ICM to external documents. Default (0.4 inches)

IMAGE.orientation

`preference` to specify image orientation.

1. = "portrait" <- default
2. = "landscape"
3. = "auto"

Default: "portrait"

IMAGE.paperSize

`preference` to specify paper size.

1. = "Letter (8.5x11")" <- default
2. = "Legal (8.5x14")"
3. = "11x17""
4. = "A4 (210x297mm)"
5. = "A3 (297x420mm)"

Default: "Letter (8.5x11")"

IMAGE.rgb2bw

`rarray` of 6 elements defining translation of rgb colors into black and white ('bw') grades. The array is {RED_scale, GREEN_scale, BLUE_scale, RED_bias, GREEN_bias, BLUE_bias} and the default values are {0.3125, 0.5, 0.1875, 0., 0., 0.}.

IMAGE.writeScale

an `integer` parameter used to increase the image resolution in the Quick Image Write tool (see a little camera on the top toolbar). This tool uses the

```
  write image png window= N * View(window)
```

command where N defines if the image is N-times bigger than the screen image. This parameter can be changed from **File/Preferences/Image** dialog.

LIBRARY

table containing string paths of the icm parameter files, which are loaded by the `read library [mmff]` command. The library files will be taken from the `s_icmhome` directory if no explicit directory is provided. Extensions are automatically added. Defaults:

```
LIBRARY.bbt="icm"  # bond bending types
LIBRARY.bci="icm"  # mmff bond charge increments
LIBRARY.bst="icm"  # bond stretching types
LIBRARY.clr="icm"  # colors, gui controls
LIBRARY.cmp="icm"  # amino-acid comparison matrix
LIBRARY.cnt="icm"  # distant restraint types
LIBRARY.cod="icm"  # atom codes
LIBRARY.hbt="icm"  # hydrogen bonding types
LIBRARY.hdt="icm"  # hydration types
LIBRARY.lps="icm.lps"  # loop database, rebuilt with write model [append]
LIBRARY.men="icm.gui"  # GUI commands. can be reloaded with 'read gui'
LIBRARY.mmbbt= "mmff"  # mmff bond bending
LIBRARY.mmbst= "mmff"  # mmff bond stretching
LIBRARY.mmtot= "mmff"  # mmff torsions
LIBRARY.mmvwt= "mmff"  # mmff van der Waals
LIBRARY.rst="icm"  # variable restraint types
LIBRARY.tor="icm"  # precomputed icmff torsion params
LIBRARY.tot="icm"  # torsion types
LIBRARY.vwt="icm"  # van der Waals types
LIBRARY.res={"icm","usr"}
```

Example:

```
LIBRARY.res=LIBRARY.res // "./benz.res" # just append
LIBRARY.cod="./newCodes.cod"
read library
```

LIBRARY.men

LIBRARY.men defines a `string` with a filename to the file with the menus.

Two possibilities:

- ♦ define LIBRARY.men in the _`startup` file for the desired menus to be activated

♦ The menus can later be extended with the `read gui` *filename* command.

```
# open the $ICMHOME/_startup file and add this line
  LIBRARY.men = Getenv("HOME")+"/.icm/icm.gui"  # now the ICM GUI will invoke your file
```

LIBRARY.res

a `string array` or file names of the `residue libraries` . File extensions can be omitted, e.g. LIBRARY.res={"icm","user","./lib/mylibrary"}

OBJECT

Controls atom requisites which are written to a file in the `write object` command. Extensions are automatically added. Defaults:

```
OBJECT.bfactor  =yes
OBJECT.charge   =yes
OBJECT.occupancy=yes
OBJECT.site     =yes
OBJECT.display  =no
OBJECT.library  =no
OBJECT.auto     =no
```

Example:

```
OBJECT.auto = no
OBJECT.display = yes
read object "crn"
display ribbon a_/1:40
set plane 2
display cpk a_/12
write object "tm"  # graphics and planes are written
delete a_*.
read object "tm"
```

PLOT

Contains settings used by the `plot` command. All real sizes are expressed in the Postscript "points" equal to 1/72" (about 1/3 mm).

PLOT.box

`rarray` of the origin and relative sizes of the ICM plot frame: { X_origin, Y_origin, X_size, Y_size }. Box `{0. 0. 1. 1.}` fits the page optimally.
Default ({0. 0. 1. 1.}).

PLOT.color

`logical` to generate a color plot. Usually it does not make sense to switch it off because your b/w printer will interpret the color postscript just fine anyway.

PLOT.font

`preference` for the title/legend font. The font size can only be redefined by editing the *.eps file (search for the number before the `scalefont` string). Available choices:

- ♦ 1 = "Times-Bold"
- ♦ 2 = "Times-Roman" <- default choice
- ♦ 3 = "Helvetica"
- ♦ 4 = "Courier"
- ♦ 5 = "Symbol"

PLOT.fontSize

`real` font size. Any reasonable number from 3. (1 mm, use a magnifying glass then) to 96.
Default (10.0).

PLOT.gridLineWidth

`real` width of grid lines. Use a small number (e.g. 0.01 for thin grid lines). Default (0.2).

PLOT.lineWidth

`real` line width for graphs (not the frame and tics)
Default (1.0).

PLOT.markSize

`real` mark size in points. Allowed mark types: line, cross, square, triangle, diamond, circle, star, dstar, bar, dot, SQUARE, TRIANGLE, DIAMOND, CIRCLE, STAR, DSTAR, BAR. Uppercase words indicate filled marks.
Default (1.0).

PLOT.numberOffset

`integer` offset for the X-coordinate with the `number` option. This option is used in a number of macros generating multi-section plots for amino-acid sequences.
Default (0).

PLOT.Yratio

`real` aspect ratio of the ICM plot frame. Using `link` option of the `plot` command is equivalent to setting this variable to 1.0. If `PLOT.Yratio` is set to 0. , the ratio will be set automatically to fill out the available box optimally.
Default (0.8).

PLOT.logo

`logical` switch for the ICM-logo on the plot.
Default (`yes`).

PLOT.orientation

`preference` for the plot orientation. Currently inactive. Default (`yes`).

PLOT.seriesLabels

`preference` to indicate position of a series/color legend inside the plot frame. You can provide individual names for each series in the optional `string array` argument of the `plot` command. (e.g. plot M_XY1Y2 {"Title","X","Y","Ser 1","Ser 2"}) Available choices:

- ♦ 1 = "none"
- ♦ 2 = "right" <- default choice
- ♦ 3 = "left"
- ♦ 4 = "top"
- ♦ 5 = "bottom"

PLOT.labelFont

`preference` for the data point label font. You can also redefine the font size with the `PLOT.fontSize` variable. Available choices:

- ♦ 1 = "Times-Bold"
- ♦ 2 = "Times-Roman" <- default choice
- ♦ 3 = "Helvetica"
- ♦ 4 = "Courier"
- ♦ 5 = "Symbol"

PLOT.rainbowStyle

`preference` defining the color spectrum used by the `plot area` command. This command lets you plot a function of 2 arguments and show the function value by color. By default the plot command uses the minimal and maximal values of the provided matrix. You can enforce the range with the `color` option. Available choices:

- ♦ 1 = "black/white"
- ♦ 2 = "blue/white/red" <- default choice
- ♦ 3 = "blue/rainbow/red"

Example:

```
 read matrix  # def.mat is the default one
 PLOT.rainbowStyle=1
 plot area def display   # grey-scale, automatic min and max
 PLOT.rainbowStyle=3
 plot area def color={-10.,0.} display  # enforce new range
 PLOT.rainbowStyle=2
 plot area def transparent={-2.,8.} display
# low values - blue, middle [-2.,8.] - invisible, large red
```

SEQUENCE.restoreOrigNames

When sequences from GenBank are read in ICM, they get a `comment` (or description) line that describes their original complicated name. The description gets this : `" Orig.name: "`*origSeqName*. This name can be restored upon writing in various formats including `write alignment fasta` if this flag is set to `yes` . By default this `logical` variable is set to `no` . Example:

```
delete a,b,c
make sequence 3 10 # creates sequences a b c
set comment a "an experimental sequence Orig.name: expr "
align a b c  # creates aln
write aln fasta "tmp" # keeps a b c, default should be no, unless you redefined it
write aln fasta "tmp" SEQUENCE.restoreOrigNames=yes  # restores the names.
```

SITE

This table contains parameters and preferences used to display the `sites`, or important residues.

SITE.appendStyle

SITE.appendStyle =

1. "none"
2. "merge source"

Allows to extend the site *description* with the name of a new sequence, even if the text of the description if the same in the `copy site` command. For example, if there is a site in two different sequences, say A_PIG A_DUCK, but they are being transferred to a third sequence with two consecutive `copy site` commands, the resulting description may indicate something like that:

- ♦ `FT 135 135 ACT_SITE active site Serine (from A_PIG)`, if the style is 1
- ♦ `FT 135 135 ACT_SITE active site Serine (from A_PIG,A_DUCK)`, if the style is 2

See also:

- ♦ `site`
- ♦ `copy site`

SITE.defSelect

string of significant site types (shown as one letter abbreviations) Sequence identity in the alignment positions which have one of those sites is additionally rewarded in the alignment score calculation.
Default: `"ABFGLMstepm"`

SITE.labelOffset

(default 5. A) the `real` offset of the site label with respect to the residue label atom.

SITE.labelStyle

the style `preference` of the displayed site information:

1. "none"
2. "symbol" # one letter symbol, see `site` .
3. "comment"
4. "full"

The `SITE.labelStyle` can also be specified locally for a given site either when one creates a site (e.g. `set site a_/2:4 "comment" label=3` . In this case zero means 'unset', or interactively by clicking on the lower-left area of the site label. One can select residues by numerical version of the local `SITE.labelStyle` preference, e.g. `a_/F2` .

SITE.labelWrap

0.5 (inactive)

SITE.showSeqSkip

the `string` of the `site` types skipped in the `show sequence` (or alignment) commands.

SITE.wrapComment

the `integer` length of the comment line. The longer lines will be automatically wrapped in the graphics view. Default: `30`

TOOLS

parameters for some ICM tools. The `TOOLS.superimpose..` parameters control the `superimpose minimize`

TOOLS.edsDir

A directory for the electron density `map` repository. If this path is empty (i.e. `TOOLS.edsDir==""`) the loaded maps from are cashed into `s_userDir` + "/data/eds/" directory

See `loadEDS` and `loadEDSweb`

TOOLS.membrane

This real array contains the geometrical parameters defining shapes with lipids or water for implicit solvation calculations. The array may contain multiple sections of 5 elements each.
Each atom is considered either buried by other explicit atoms of the objects or exposed. Five types of environment are possible depending on the `surfaceMethod` and the "sf" term:

- ♦ vacuum (`set term "sf" off`)

- ♦ constant tension (`surfaceMethod=2; set term "sf"; surfaceTension=0.02`)
- ♦ water (`surfaceMethod=2;set term "sf"`, exposed atom solvation density in col3 of `icm.hdt`)
- ♦ apolar (`surfaceMethod=3;set term "sf"`, exposed atom solvation density in col4 `icm.hdt`)
- ♦ membrane (`surfaceMethod=3;set term "sf"`, exposed atom solvation density in col3 or col7)

The surfaceMethod is defined as `membrane` then each atom can be either in water environment or in lipid environment. The attribution is done according to the atom coordinates and a series of shapes coded by the TOOLS.membrane array.

Each shape is coded by 5 numbers in the TOOLS.membrane array. The following shapes with lipid-like environment can be introduced.

Name	Type(e1)	e2	e3	e4	e5	Description	Examples
membrane	1	z_bott	z_top	z_bmargin	z_tmargin	Z-membrane with transitional layers	{1.,0.,30.,5.,5.}
z-chimney	2	x1	y1	x2	y2	infinite rectangular beam in Z direction	{2.,-5.,-5.,5.,5.}
z-cylinder	3	xct	yct	radius	0.	infinite round cylinder in Z direction	{3.,7.,7.,15.,0.}
cube	4	xct	yct	zct	half-edge-length	cube with defined center and half-size	{4.,0.,0.,0.,10.}
sphere	5	xct	yct	zct	radius	sphere with defined center and radius	{5.,0.,0.,0.,10.}

Each shape only defines the environment inside itself and is applied sequentially. To define *water* environment inside *preceding* lipid shape, use the **negative** type, e.g. {..lipid_shape.., -4.,0.,0.,0.,3.} for a water cube of half size 3.. Examples:

```
TOOLS.membrane = {1.,0.,30.,5.,5.} # one membrane from z=0 to z=30A with 5A transitiona
TOOLS.membrane = {5.,0.,0.,0.,4.,  5.,10.,10.,10.,4.} # two lipid spheres at 0,0,0 and
TOOLS.membrane = {5.,0.,0.,0.,6., -5.,0.,0.,0.,3.}    # a spherical layer, two concentr
```

See also: `icm.hdt` , `surfaceMethod` = 4 , energy term `"sf"`

TOOLS.minSphereCubeSize (default = 5.)

This parameter adjusts the minimal size of a cube for fast scanning of interatomic pairs. These calculation may be necessary when pairs of atoms are processes during an energy calculation or in a `Sphere` function. If an odd atom or group of atoms is far away from the rest of the object (an unusual and undesired situation), an exceedingly large number of cubes between the groups need to be created. In this case the minimal size of an edge needs to be increased to deal with those sparse atoms. If you see a warning involving this parameter, have a look at the coordinates of your object and make sure that the atom positions are correct.

TOOLS.pdbReadNmrModels

TOOLS.pdbReadNmrModels = "first"

1. = `"first"` : reads only one model from a multi-model (e.g. NMR) pdb file
2. = `"all"` : reads all models from a multi-model (e.g. NMR) pdb file and creates a separate object for each of them
3. = `"all stack"` : creates one object and loads all other models as a stored cartesian stack

This preference is set to `"first"` by default. Resetting it to `"all"`is equivalent to option `all` in `read pdb` . Setting the preference to 3 is equivalent to the `read pdb all stack` *s_pdbMultiModelFile* command. Example:

```
TOOLS.pdbReadNmrModels = "all"
read pdb "1dkc"  # all 10 models are read in as a_1dkc_1., .., a_1dkc_10.
read pdb "1dkc" TOOLS.pdbReadNmrModels=3 # one object with a built-in stack is created
```

TOOLS.smilesXyzSeparator

ICM has an option to generate smiles with coordinates which is a much more compact (and one line only) replacement for an `mol` / `.sdf` file

TOOLS.superimposeMaxIterations

The maximal number of iterative superpositions with gradually improved weights in the `superimpose minimize` procedure that optimizes the weighted rmsd to find the best superposition core. Do not afraid to set this number to a very large one since the procedure will exit earlier if the convergence is achieved.

Default: 10

TOOLS.superimposeMinAtomFraction

The minimal fraction of equivalent atom pairs that will be superimposed with significant weights in the `superimpose minimize` procedure.

Default: 0.5

TOOLS.superimposeMaxDeviation

Determines the maximal atom deviation for determining the *core* subset of atoms for which the unweighted RMSD is reported in `r_2out` in the `superimpose minimize` procedure. The unweighted rmsd for a subset must be lower than this parameter.

Default: 2.0

TOOLS.tsToleranceRadius

radius around atoms where the deviations from the target are not penalized

Default: (0.)

See also: `selftether` , `term ts` , `TOOLS.tsShape`

TOOLS.tsShape and TOOLS.tsShapeData

This preference and a `rarray` of parameters supports positional restraints for all non-virtual atoms of the current object. It can be used concurrently with atom-specific selftethers and does not need the `set selftether` *as* command. The `TOOLS.tsShape` preference has the following values:

```
  1 = "none" <-- current choice
  2 = "sphere"  # also a spherical layer if two radii are specified
  3 = "box"
```

The `TOOLS.tsShapeData` real array contains the parameters needed for a shape restraint. Note that while TOOLS.tsShape preference can be specified as an inline argument of a `minimize` or `montecarlo` command, the array can not. Therefore the values need to be filled before you call the command, e.g.

```
TOOLS.tsShapeData = {20.}  # radius of the sphere
minimize "vw,ts" TOOLS.tsShape=2
```

sphere. For the sphere option the radii and the center x,y,z parameters can be specified.

TOOLS.tsShapeData = { [*minimal_radius*] *max_radius* [*x y z*] }

The radii can be provided in any order. The default radius is 20. and the center of the spherical restraint is the origin ({0. 0. 0.}

Examples:

```
 TOOLS.tsShapeData = {10.} # keep atoms inside R=10
 TOOLS.tsShapeData = {10. 10.} # keep atoms on the surface of the sphere R=10
 TOOLS.tsShapeData = {10. 15.} # penalty free is the spherical layer between R=10 and 1
 TOOLS.tsShapeData = {10. 15. 3. 3. 3.} # layer between R=10 and 15 around  3.//3.//3.
 TOOLS.tsShapeData = {10.}//Mean(Xyz(a_//!vt*)) # do not let molecule fly beyond 10A fr
```

box.

TOOLS.tsShapeData = $\{x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}\}$

These six parameters are compatible with the Box function. Example:

```
TOOLS.tsShapeData = Box( a_// 3. )
#
TOOLS.tsShapeData = 100.//100.//-1.//100.//100.//1.  # keep atoms in 1A layer around Z-
```

See also: selftether , term ts

TOOLS.tsWeight

Weight for a special terms "ts" that can be used in minimize to tether the atoms to its initial set of coordinates (see selftether). To keep only some atoms self-tethered, use option selftether= *as_selfTetheredAtoms* of the minimize command. The convert command and set selftether set them.

Example:

```
  build string "lys"
  randomize v_//x*
  minimize "vw,to,ts"  selftether=a_//ca,c,n  TOOLS.tsWeight=10.
```

Default: (0.)

See also: term ts , selftether , TOOLS.tsShape.

TOOLS.writePdbRenameRes

Defines translation rules for the internal ICM residue names in show pdb or write pdb commands. For example, names ra for adenosine, or cyss for the bonded cysteins, can be translated to other names during the pdb export into ade or cyx . The format is the following: *icm_res_name* **,** *exported_pdb_res_name* **;** ... , e.g.

```
 TOOLS.writePdbRenameRes = "cyss,cyx;his,hid"  # will rename two residues
```

See also: show pdb write pdb .

WEBLINK

This table contains definitions of types of web links used in the web, show html, and write html commands. The table is read from "WEBLINK.tab" file from the $ICMHOME directory. Change this file for your own definitions. The weblink specification is used to extend the argument string substituted for %s (e.g. "IL2_HUMAN" element of the table array linked according to the type
SP %s "http://www.aaa?%s" will be transformed into the <a href="http://www.aaa?IL2_HUMAN">IL2_HUMAN</a> link. If %Ns specification is used, only N characters of the argument string will be retain in the link. For example,
PDB %s "http://www.pdb?%4s" and 1xyz_a15_25 (specifying chain and residue range) will be translated into

`<a href="http://www.pdb?1xyz">1xyz</a>_a15_25` which in your browser will look like this:

"AUTO" is another type which can be used in the link S_ "TYPE" ... expression. In this case the DB type is automatically recognized according the database reference string pattern (see also `WEBAUTOLINK`). An example table:

```
#>T WEBLINK
#>-DB------DR--LINK-------
 PDB     %s  http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&amp;form=6&amp;u
 NCBI    g%s http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&amp;form=6&amp;u
 EMBL    %s  http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&amp;form=6&amp;u
 SP      %s  http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&amp;form=6&amp;u
 SPA     %s  http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=s&amp;form=6&amp;u
 PROSITE %s  http://saturn.med.nyu.edu/srs/srsc?[PROSITE-acc:%s]
 MED     %s  http://www3.ncbi.nlm.nih.gov/htbin-post/Entrez/query?db=m&amp;form=6&amp;u
```

Example:

```
 read table "seqcomp.tab"  #contains references to different databases
 web SR link SR.NA1 "PDB" SR.NA2 "AUTO"
```

WEBAUTOLINK

This table contains definitions of web link string patterns for automatic recognition in the `web`, `show html`, and `write html` commands. The table is read from the "WEBLINK.tab" file in the `$ICMHOME` directory. Change the file for your own definitions. Recognition is not perfect because the patterns overlap.
Example:

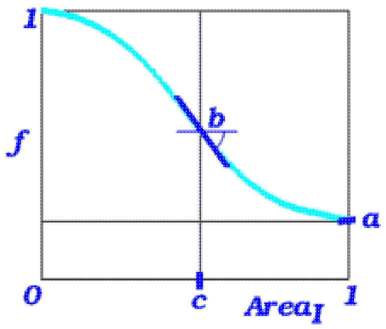
```
 read table "seqcomp.tab"  #contains references to different databases
 web SR link SR.NA1 "AUTO" SR.NA2 "AUTO"
```

Other shell variables

defCell

the `real array` of the default cell parameters. This definition is used in the `Resolution` and `MaxHKL` functions if cell parameters are not provided as arguments.
Default: {1. 1. 1. 90. 90. 90.}

accFunction



the `real array` of the `solvent accessibility` penalty parameters (as described in `Batalov and Abagyan, 1999`).
It contains the values of *a, b, c* and *E* damping parameters for amino acid substitution scores. Generally, if a residue is completely buried (*Area=0)*, its substitution scores will be used without changes. If it is completely exposed, its substitution scores will be multiplied by the minimal possible value of *a*. Between these cases the substitution scores are modulated by a smooth ("arctangent") function with a saddle point at *Area=c*, where the slope will be *-b*. The fourth parameter is reserved for development.

This definition is effectively implemented in the `Align(` *seq_1 seq_2* `area`) }, `Score` functions

and find database command.
Default: {0.33 2.35 0.211, -15.0}.
See also: alignMethod .

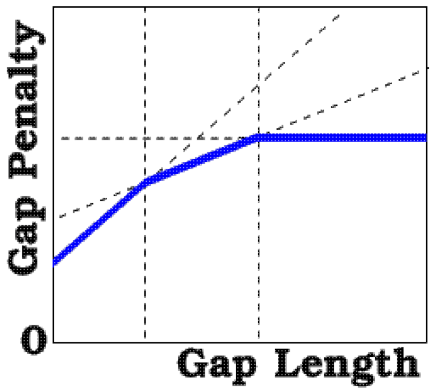
gapFunction

the real array of the gap penalty parameters, which represent a piecewise-linear concave function (as described in Batalov and Abagyan, 1999).
ATTENTION: at the present time this gapFunction is only active when alignMethod =2. The first two values replace gapOpen and gapExtension traditional values. If present, the third element of the array represents the length of the gap, starting at which further gapExtensions become equal to the fourth element of the array. Likewise, if more elements are present, they represent pairs of the threshold lengths of the gap and the new **gapExtensions** values. For example,

```
 gapFunction = {2.4 0.15 10. 0.05 20. 0.}
```

means that

- gap penalty=2.25+0.15*L for L={ 0..10} (and for L=1 it is 2.4= gapOpen),
- gap penalty=3.25+0.05*L for L={11..20} and
- gap penalty=4.25 for L>20



The calculations are fastest for the traditional two-element gapFunction. The three- or four-element gapFunction invokes the optimized routines and is 50-70% slower. The general kind gapFunction costs approximately 70-90% additional time for every pair of gapFunction values. If the last gapExtension is zero, it may be omitted. This definition is effectively implemented in the Align , Score functions and find database search command.

Default: {2.4 0.15}.
Recommended (put it in your _startup file): gapFunction = {2.4 0.15 10.}
This set will produce fast and structure-like alignments.
See also alignMethod, and accFunction (the accessibility attenuation parameters).

I_out

an integer array in which the output of some commands is stored.

M_out

matrix in which the output of some commands is stored.

R_out

real array in which the output of some commands is stored.

Functions returning in R_out:

- Axis # middle point of the axis
- Disgeo # returns error sum of negative scaled eigen values in R[0], and first three 3 scaled eigen values

- ♦ `LinearFit` # residuals
- ♦ `Xyz(` ..) returns inverse transformation
- ♦ `learn` returns model accuracy and stds.

R_2out

auxiliary `rarray`. Used in addition to `R_out`.

S_out

`string array` in which the output of some commands is stored.

swissFields

`string array` of SWISS-PROT fields to be read by default in `read sequence swiss`

If the field name starts from a minus ('-'), this field will be ignored in the feature table list.

Example:

```
swissFields={"-HELIX ","-COIL ","-STRAND","-TURN "}
# to suppress the FT records with the secondary structure info
```

readMolNames

`string array` in which the SDF-file comment fields containing database compound identifier and description are preset. There is a standard place where database compound identifier should be stored in SDF (MOL)-files. This is the first line of the entry. However most of the database providers got used to leaving this line empty. Instead they put identifier and description in the end of the file in the following fashion:

```
...
M  END
> <CAT_NO>
R150002

> <NAME>
(5-OXO-HEXAHYDRO-PYRANO[3,2-B]PYRROL-1-YL)-ACETIC ACID METHYL ESTER

$$$$
```

In this particular case before using such database set

```
readMolNames = {"<CAT_NO>" "<NAME>"} # useful for Sigma-Aldrich files
```

Another example:

```
readMolNames = {"<CODE>" "<IUPAC_NAME>"} # useful for ACD database
```

Named Atom/Residue/Molecule/Object/Variable Selections

Selections of atoms, residues, molecules, objects or internal variables (torsions, planar angles, bond lengths) can be stored in variables.

Examples:

```
 cc = a_//ca                 # created named selection variable cc
 show cc & a_/3:15           # use it in the expression
```

In this case the named selection **cc** is a true ICM-shell variable, not just an alias for the Ca selection. Please do not confuse it with another useful mechanism which allows you to use a **string** in a selection. This mechanism is used in scripts and macros.

Example:

```
 cc = "a_//ca"      # in this case cc is a string, not a selection
 show $cc & a_/3:15 # $cc is replaced by a_//ca before parsing
```

How to store and exchange selections in strings: Examples of using the `String` (*os|ms|rs* [name|number]) function to return a residue selection:

```
l_showResCodeInSelection = yes # the default
res_str = String( Res(Sphere( a_H [1] a_A//!h*,ca,c,n,o 3.5 )))  # same as with option
show res_str
 a_2c0cb.b/^T159,^S205,^K209,^Y224,^V248,^Y275,^L305,^M356,^N361
# or
l_showResCodeInSelection = no
res_str = String( Res(Sphere( a_H [1] a_A//!h*,ca,c,n,o 3.5 )))  # same as with option
show res_str
 a_2c0cb.b/159,205,209,224,248,275,305,356,361
```

However, be careful with using it in an arbitrary case at the atomic level since it may lead to a string that is too long.

as_out

an atom/residue/molecule/object `selection` variable where some commands or functions store their output:

- `Rmsd`
- `Srmsd`
- `superimpose`
- `set tether`
- `show drestraint`
- `show tethers`
- `find chemical`

If atoms a_1./3/ca,c,n relate to atoms a_2./45/ca,c,n, then the first set will end up in `as_out` and the second in `as2_out`.

as2_out

the second set of atoms (`selection`) returned by the following commands and functions:

- `Rmsd`
- `Srmsd`
- `superimpose`
- `set tether`
- `show drestraint`
- `show tethers`

See also: `as_out`.

vs_out

The variable selection where some commands or functions store their output:

- `read variable` saves a selection of loaded variables;

Chemical arrays and tables. Operations, virtual chemistry.

Chemical arrays can be read from multiple external formats (e.g. `.sdf` , `ml2`) either as standalone arrays, or columns in chemical spreadsheets. These arrays are abbreviated as `X_`

Reading/Writing Chemical Arrays and Tables

`read table mol` reads chemical table from file

`write table mol` writes chemical table to file

InChI conversion

`converts chemicals to InChI or InChI_key` `converts InChI to chemical array`

Smiles conversion

`converts smiles to chemical array`

`converts chemical array to smiles`

Conversion from loaded 3D objects

`converts 3D selection to chemical array`

`converts chemical to 3D`

`converts chemical to 3D and optimize`

`converts loaded object to 3D and optimize`

`converts loaded object to 3D ICM object, preserve coordinates`

Modifying chemicals

`modify chemical` perform chemical group modifications on chemical arrays

`apply 2D depiction`

Generating chemicals. Virtual chemistry

`Markush library generation`

`Chemical reactions`

Comparing, Searching and Chemical Matching

`searching in chemical tables`

`search in Molcart databases`

`search in loaded 3D objects`

`chemical distance/similarity`

Other chemical search related functions

`Nof chemical`

`Index chemical`

Predicting chemical properties

`Predict`

Chemical superposistion

`superimpose` `Rmsd` `Srmsd`

`chemSuper3D`

Pharmacophore analysis

`pharmacophore search`

`superimpose`

`Rmsd`

```
create pharmacophore
```

Batch chemical processing,

```
_chemBatch
```

SMILES and SMARTS

Simplified Molecular Input Line Entry Specification which stems from traditional string notation of graphs and trees, e.g. the Newick notation. The acronym introduced by David Weininger to represent chemical valence model by a string (e.g. CC=O). It can also be used as an exchange format for chemical data. The algorithm was `published in 1988` and is described in detail at the WWW site of Daylight Chemical Information Systems, Inc. `http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html` . Another description can also be found here: `http://en.wikipedia.org/wiki/Simplified_molecular_input_line_entry_speci`

The SMILES notation allows one to represent a 2D chemical drawing as a string, (e.g. `"C1CCCCC1"` for cyclohexane). The SMARTS notation is an extension of SMILES that allows one to specify chemical patterns with wildcards for atoms or bonds, e.g. `" [C`,N,O]?" . **SMARTS**SMARTS is an extension of the SMILES notation to include wildcards. This chemical patterns can be used in chemical queries and is described here: `http://www.daylight.com/dayhtml/doc/theory/theory.smarts.html`

The primitives supported in ICM include the following (note that the **atom** primitives in general are in brackets, e.g. `[Cl]` for a chlorine atom):

Symbol	Description	Examples
`*`	any atom	`*`
`a`	aromatic	`aN(=O)O`
`A`	aliphatic	`AAA`
`C`	aliphatic carbon	
`c`	aromatic carbon	
`[#6]`	any carbon	
D*n*	the number of heavy neighbors	`[*;D2]` any atom with two non-H connections
H*n*	number of attached hydrogens	`[*;H2]` atom with two hydrogens (see also `Y`)
R*n*	the number of rings the atom belongs to	`[#6;R2]` any carbon in two rings
r*n*	the size of smallest ring the atom belongs to	`[*;r6]`
v*n*	valence, sum of bond orders of all neighbors	
X*n*	the number of all neighbors including hydrogens	
`-`	negative charge	[--], [-2]
+*n*	positive charge	[++], [+2]
^*n*	sp1,sp2,sp3 hybridization	e.g. [C;^2] sp2 carbon # ICM extension
y*n*	ring number in SSSR	e.g. [*;y1] any atom which belongs to the first ring # ICM extension
Y*n*	number of at least attached hydrogens	`[*;Y2]` atom with two or more hydrogens # ICM extension
#*n*	atomic number	`[#7]`
@	anticlockwise chirality	`C[C@H](F)O`
@@	clockwise chirality	
~	any bond	`C~C`
:	aromatic bond	`c:c`
-,=,#	single, double and triple bonds	C#C
=&!@	bond SMART notation for double, not in ring	acC=&!@Cca

!*primitive*	negation	[!C] non-aliphatic carbon, [*;!R] any atom not in a ring
*expr1***&***expr2*	logical and (high precedence)	[c,n&H1] any arom carbon OR H-pyrrole nitrogen
*expr1***,***expr2*	logical or	[C,N,O] C or N or O
*expr1***;***expr2*	logical and (low precedence)	[c,n;H1] arom carbon OR nitrogen with one hydrogen

Aromatic vs aliphatic

Note that uppercase atoms in SMARTS will only match aliphatic (not aromatic) atoms. For example "C" will not match any atom in "c1ccccc1" (or "C1=CC=CC=C1") ring. If you want to match both cases you should use [#] notation. For example "[#6]" will match both aliphatic and aromatic carbons.

Recursive SMARTS

This SMARTS feature allows you to define "atomic enviroment" when matching. "Enviroment" atoms will not be included into result match.

For example `[C&!$(C=O)&!$(C#N)]` will match any aliphatic carbon not double bonded to an oxygen and not triple bonded to a nitrogen.

Example:

```
build smiles "CN(CCN(CC(C=CC(=C1)C(=O)NC(C=CC(C)=C2NC(N=CC=C3C(C=CC=N4)=C4)=N3)=C2)=C1)
display xstick
find chemical a_ "[O,S&v2,N&^2&X2,N&^1&X1,N&^3&X3]" all # find hydrogen bond acceptors
color xstick as_out rgb = { 152 251 152 }
find chemical a_ "[!#6;!H0]" all # find hydrogen bond donors
color xstick as_out rgb = { 238 130 238 }
find chemical a_ "a" all        # aromatic
color xstick as_out rgb = { 255 165 0 }
find chemical a_ "[C&!$(C=O)&!$(C#N),S&^3,#17,#15,#35,#53]" all # hydrophobic
color xstick as_out rgb = { 224 255 255 }
```

R-groups, attachment points, and chemical searches.

In reactions, Markush structures and building blocks two additional wildcards are used:

Group	Example
`R1`,--R2,.. groups	[R1]N(=O)O
`[C*]` attachment points	

Do not confuse *any atom*, e.g. [*], and *attachment point* where the asterisk follows an atom symbol in square brackets, e.g. [C*]. Example in which an attachment point is added to a carbon attached to a ring:

```
Replace( Chemical( "C(=CC=CC1)C=1C" ), "Cc(:c):c" "[C*]c(c)c" exact )
```

See also:

- `Smiles` function and the `build smiles` command.
- `Nof(` *chemarray s_smart* `)` # the number of found fragments
- `Index(` *chemarray s_smart* `)`
- `Replace(` *chemarray s_smartFrom s_smartReplace* [`exact`])
- `modify` *chemarray s_smartFrom s_smartTo*
- `find molcart table=` *s_tableName s_smart* [`name=` *s_outputTable*]

SMARTS chemical expression language, an extention of the `smiles` language.

SOAP services and communications

To access some external services there is a protocol called SOAP. Now ICM can send a SOAP request and get the result back to ICM.

Sending request to the SOAP server

A SOAP request is a special XML text which contains :

- the SOAP method name and a name-space
- the method arguments

In ICM you can form a SOAP message using the `SoapMessage` function. It creates a special *soapMessage* object which holds SOAP method name and it's arguments.

Example:

```
# create a message with SOAP method and a namespace
req = SoapMessage( "doSpellingSuggestion","urn:GoogleSearch" )
# add method arguments
req = SoapMessage( req,  "key","btnHoYxQFHKZvePMa/onfB2tXKBJisej" ) # get key from goog
req = SoapMessage( req,  "pharse", "Bretney Spers" )  # some misspelled pharse
```

Once the message is ready it can be send to the server using the `read http` command.

`read string` *s_soapServiceURL* + " " + `String(` *soapMessage*)

The result of the server response will be stored into `s_out` variable. It can be parsed to a soapMessage object using the `SoapMessage` function.

Example:

```
HTTP.postContentType = "text/xml"
read string "http://api.google.com/search/beta2" + " " + String(req)
res = SoapMessage( s_out )
if Error(res) == "" then
 # process message
endif
```

Processing SOAP results

To access the content of SOAP message, the `Value` function is applied

If the result of the SOAP response is a simple value, such as `integer`,`real,`string,`sarray,`rarray,`iarray, then it will be automatically casted to the corresponding ICM type. Otherwise a special type of `parray` will be returned.

In some cases the result returned by SOAP server is actually some complex data structure (not just a single string or number) The most common complex SOAP types are 'struct' and 'array'. Each of them can either contain a simple type of other 'struct' or 'array'.

You may navigate through this structure using index expressions:

soapObject[*i_integerIndex*]

or

soapObject[*s_stringIndex*]

The number of elements in the array of struct can be returned by `Nof` function.

`Sarray` of field names of a struct can be returned by `Name` function

For example the following code navigates through the result obtained from the google search service.

```
  res = Value( SoapMessage( s_out ) )
  s_html = ""
  s_html += "Searched web for <b>" + res["searchQuery"] + "</b>. Search took " + res["s

  elements = res["resultElements"]
  for i=1,Nof(elements)

    resElement = elements[i]

    s_html += "<br>"

    cat     = String( resElement["directoryCategory"]["fullViewableName"] )
    summary = String( resElement["summary"] )
    title   = String( resElement["title"] )
    snippet = String( resElement["snippet"] )
    url     = String( resElement["URL"] )
    cachedSize     = String( resElement["cachedSize"] )

    if (Length(title) != 0) then
        s_html += "<font color=\"#0000FF\"><b><u>" + title + "</u></b></font><br>"
    else
        s_html += "<font color=\"#0000FF\"><b><u>" + url + "</u></b></font><br>"
    endif

    if (Length(snippet) != 0) s_html += snippet + "<br>"
    if (Length(summary) != 0) s_html += "<font color=\"#808080\">Description:</font> "
    if (Length(cat) != 0) s_html += "<font color=\"#808080\">Category: <u>" + cat + "</
    if (Length(title) != 0) s_html += "<font color=\"#008000\"><u>" + url + "</u> - " +

  endfor
```

{ KEGG database }

KEGG (Kyoto Encyclopedia of Genes and Genomes) is a database resource that integrates genomic, chemical, and systemic functional information. In particular, gene catalogs in the completely sequenced genomes are linked to higher-level systemic functions of the cell, the organism, and the ecosystem.

Example of few requests:

```
l_info = l_commands = no
HTTP.postContentType = "text/xml"
HTTP.soapAction = "SOAP/KEGG"
url = "http://soap.genome.jp/keggapi/request_v6.2.cgi"

req=SoapMessage( "get_pathways_by_genes" "SOAP/KEGG" )
req=SoapMessage( req "genes_id_list", {"hsa:5292"} )
read string url+" "+String(req)

sss = SoapMessage( s_out )

if (Error(sss) == "") then
  S_path = Value(sss)
  print "pathways = " Sum(S_path,",")

  for i=1,Nof(S_path)
    req=SoapMessage( "get_references_by_pathway" "SOAP/KEGG" )
    req=SoapMessage( req "pathway_id", S_path[i] )
    read string url+" "+String(req)
    sss = SoapMessage( s_out )
    print "references for " S_path[i]
    Value(sss)
  endfor

endif
```

For KEGG WSDL file click `here`. API is defined `here`

Related functions: `SoapMessage` `Value` `Error` `Nof` `Type` `Name`

Creating your own GUI elements: Programming GUI.

There are three possibilities to add a new menu or a popup dialog.

1. **Main menus**. Add or modify a section in the `icm.gui` file. To program the main GUI menus and popup dialogs, you need to modify a single file, called `icm.gui` . This file resides in the `$ICMHOME` directory. The icm.gui file contains controls for each menu item. It the menu item requires a dialog, it is automatically generated.
2. **Popup dialogs in tables**. Add a header element called `cellPopup` or `tablePopup` need to be added to a table, e.g.

   ```
   add column t {1 2 3}
   add header t "POPT New Table Item\nSYNT # T_Table\nSYNT # r_Enter_Real (2.)\nSYN
   add header t "POPT New Table Cell Item\nSYNT # r_Enter_Real (2.)\nSYNT print $1
   ```

   The following shortcuts are allowed:

   ```
    %# is the line number, e.g. t.mol[%#]
    %^ is the column name, e.g.
    %@ is the table name, e.g. %@.%^ is table.column
   ```

3. **Popup dialogs in internal html-documents**. For those you need to add ths following line:

   ```
   #dialog{"YourCaption"}
   ```

   so that the dialog looks like this:

   ```
   #dialog{"Enter name"}
   # s_name ("Kosmo")
   # ff_file|*.html ("")
   print $1 $2

   #dialog{ "Sample Dialog" }
   # txw_Enter_Text ()
   txt = %s_out   # s_out is not a safe place (might be overwritten)
   print Length(txt)
   ```

   Notice underscore containing names like `s_` need to be capped with the percent symbol

The `icm.gui` section consists of the following fields:

Field	Example	Description
MENU	Display.Advanced.Pockets	this menu item will be added in the top menu bar
OPTN	Apply	optional. Allows to specify special kinds of dialogs.
OPTN	Conditions and Flags	
COMM	Comment	
ICON	icon name	
SYNT	display g_pocket	the ICM-command executed upon pressing 'OK' or 'Apply'.

An example:

```
MENU Display.Color.White
SYNT color white
```

Many examples of how to program controls for the script arguments can be found in the `icm.gui` file.

Creating controls for the main input data types.

To create a control for a particular data type one needs to add the following line.

SYNT # *t_name* (*default1* | *default2*) [*OPTIONS*>]
Example:

```
SYNT # i_enterInt (10)
```

The following input types are possible:

type	example	comment	
b_button	SYNT # b_Undisplay_distances (delete g_distances)	executes the command in () when pressed.	
d_directory	SYNT # d_PDB_Directory (SYS:s_pdbDir)	gives you a dialog to select a directory	
ff_fileDialog	SYNT # f_Input_file *.mol;*.SD*;*.sd* ()	generates a **secondary** file dialog	
f_fileDialog	SYNT # f_Input_file *.mol;*.SD*;*.sd* ()	generates a file dialog first, ignores other arguments	
g_grobMesh	SYNT # g_Mesh (*)	star means show the current list of grobs in icm	
i_integer	SYNT # i_size (10)	also allows one to filter the input, e.g. SYNT #20 i_Angle	MIN:-360 MAX:360)

		(360
l_logical	SYNT # l_high_res (yes)	
o_radioButtons	SYNT {o_icmWord1_text_1 o_icmWord2_text_2 ..}	
os1_molObjects, ms1_molecules, rs1_residues, as1_atoms		
p_preference	SYNT # p_ribbonStyle_Ribbon_Style	internal name followed by the control title.
r_realVar	SYNT # r_temperature (200.)	
R_realArray		
s_stringVar	SYNT # s_name (John)	returns quoted `"John"`
txw_Big_Text_Box	SYNT # txw_Description ($s_out)	stores edited text in s_out variable. Use %s_out to access the value.
txt_text	SYNT # txt_boxName (Text to be displayed)	free comment
u_unknown		
T_table	SYNT # T_Table_Name (*)	an asterisk (*) will be substituted with the list of tables
X_chemicalTable		
C_column	SYNT # C_Select_Column [TABLE:2,SARRAY]	selected a column of the existing table. Works in pair with T_ or X_
ch_Compound	SYNT # ch_Draw_Compound	draw compound or type SMILES/SMARTS string

Possible values of *OPTIONS* are:

option name	description	example
IARRAY,SARRAY,RARRAY,CHEMARRAY	narrow the choice for column selector C_	
RDONLY	sets control to read-only mode. User can not pick from the list.	
REQUIRED	insist that the value is not empty	
RESIZE	allow resizing of the window	`[RESIZE]`
RIGHT,LEFT,CENTER	alignment of the control on the dialog	
TABLE:~~i_tableReferenceNumber	column selector C_ need a reference to a parent table. This number is found counting *prefix_* before the table	
SMARTS	generate SMARTS expression from ch_ input	
NOCOORDS	do not include 2D coordinates into result of ch_ input	

The *i_tableReferenceNumber* needs some clarification.

```
SYNT # T_Table_A (*)  [RDONLY,LMINWIDTH:110] C_by_Column () [TABLE:1,RESIZE,LMINWIDTH:1
SYNT # { o__inner o_left o_right } [CENTER]
SYNT # T_Table_B (*) [RDONLY,LMINWIDTH:110] C_by_Column () [TABLE:6,RESIZE,LMINWIDTH:11
```

Here column selection (`C_by_Column`) for table `B` (the 3rd line) is refering to table `B` . `T_Table_B` has its reference number as `6` (not 4) because this *i_tableReferenceNumber* counter counts each `o___inner o_left o_right` as 3 words in contrast to the dollar-argument number

.

Conditional options:

VISIBLE *s_condition*| shows/hides control depending on condition	
OPTN *s_condition*	enables/disables control depending on condition

Example of "Extract Ligand" dialog :

```
MENU Extract Ligand(s)
COMM Extract ligand into chemical table
OPTN !isMini !isPharm
ICON bipm_chemring
SYNT #1 ms1_
SYNT #2 { o_"2D"_as_2D_drawing o_"3D"_keep_3D_coordinates }
SYNT #3 l_append_To_Existing_Table (no) [OPTN:(nChemTable)] X_ (*) [RDONLY,OPTN:($4)]
SYNT if (   $3 ) extractLigand $1 $2 $3 Name( $4 table )
SYNT if ( ! $3 ) extractLigand $1 $2 $3 ""
```

Example of "Merge Two Sets" dialog:

```
MENU Chemistry.Merge Two Sets
OPTN nTable>1
SYNT # T_Table_A (*)  [RDONLY,LMINWIDTH:110] C_by_Column () [TABLE:1,RESIZE,LMINWIDTH:1
SYNT # { o__inner o_left o_right } [CENTER]
SYNT # T_Table_B (*) [RDONLY,LMINWIDTH:110] C_by_Column () [TABLE:6,RESIZE,LMINWIDTH:11
SYNT # txt_Hint (inner - only molecules present in BOTH A and B tables are kept<br>left
SYNT join $2 $5 $3
```

Specifying the default values of the arguments

The general syntax of the default value specification is:

(*value1* | *value2* | *...*)

Where *default_value* can also be one of the following:

- asterisk, * : lists all values/variables of specified class, e.g. `seq_` (*)
- dynamic condition: expression in extra parenthesis containing filtering condition, e.g. ((isAmino))
- custom text/string. This is either any text or expression in the form: *internal_text@Human_Readable_Text* , e.g. `(10|0@all)` you will see `all` , but value 0 will be returned.

Examples:

```
os1_Select_Object (*)             # lists all objects loaded in current session
os1_SelectObject ((isIcmObj))        # lists all ICM objects
os1_SelectObject (*|Obj(as_graph)@Graphic Selection)        # lists all ICM objects plu

ms1_Select_Molecule ((!isWater))  # will forms a list of all molecules excluding waters
ms1_Select_Molecule ((isAmino)|(isNucl))  # will forms a list of all amino and nucl cha
ms1_Select_Molecule ((isHet)|(isAmino&nResInMol<10))   # list all ligands plus short am
```

This is a list of internal GUI functions that can be used in dynamic conditions

Functions for the Molecular arguments (`ms1_` ..):

- isAmino (lists amino chains)
- isNucl (lists nucleotide chains)
- isHet (lists hetero molecules - ligands)
- isMetal (lists metals)
- isWater (lists water)
- isDsMol (lists displayed molecules)
- isIcmObj (only molecules of ICM objects)
- isLinkedToAli (lists linked to an alignment molecules)
- hasSwissID (lists molecules with swiss ID)
- nResInMol (counts number of residues in molecule)
- nAtomInMol (counts number of residues in molecule)

Functions for the Object arguments (`os1_` ..):

- ♦ hasAmino (lists objects with amino chains)
- ♦ hasNucl (lists objects with nucleotide chains)
- ♦ hasHet (lists objects with ligands)
- ♦ hasMetal (lists objects with metals)
- ♦ hasStack (lists objects with attached conformation stack)
- ♦ nMolInObj (counts number of molecules in the object)
- ♦ nResInObj (counts number of residues in the object)
- ♦ nAtomInObj (counts number of atoms in the object)
- ♦ isIcmObj (lists ICM objects)
- ♦ isDsSelObj (lists displayed and selected objects)
- ♦ isDsObj (lists displayed objects)
- ♦ isPharm (lists pharmacophore objects)

Suppressing the automatic interpretation of X_ with the % symbol.

Note, that the GUI dialogs and controls are generated automatically every time the program finds a letter followed by underscore. For that reason if you want to **suppress** that action, use the **percent** symbol, e.g.

```
SYNT print  %s_out # do not want to generate a text dialog.
SYNT display %as_graph # here you do not want to generate an as_ selection dialog
SYNT if Nof(g_distances)==0 delete  %g_distances
```

Specifying MIN and MAX values, or adding other controls.

```
SYNT #20 i_Angle        (360|MIN:-360|MAX:360)
```

Specifying objects, molecules, residues and atoms

More detailed examples.

```
SYNT # f_Input_file|*.mol;*.SD*;*.sd* () # generates a file dialog
SYNT # r_Sphere ( 3. ) # here r_ is the type; 3. is the default value
SYNT # r_Resolution_Increase (1.2|1.5|1.75|2|2.5|3) # you can provide popular choices
SYNT # r_Ribbon_Width   (SYS:GRAPHICS.ribbonWidth) # that is how a system variable is r
```

The actual icm commands that use those arguments refer to them as `$1 $2` .. , e.g.

```
# choose a mesh and save it to a file
SYNT # g_mesh (*)
SYNT # f_file
SYNG write $1 $2
```

Layout issues

Layout defines how various input elements are arranged on the page. ICM arranges all controls in a single line horizontally. Each new line in the dialog definition starts a new horizontal layout. In some cases you need to arrange controls as a grid. *BEGINGRID* and *ENDGRID* options should be used to do this:

Example:

```
# l_Anilines (yes) [BEGINFRAME:Options,BEGINGRID] l_Nitro (yes)
# l_Metals (yes) l_Carboxylic_Acid (yes)
# l_Hologen_Hydrides (yes) l_Aldehydes (yes)
# l_Alkyl_halides (yes) l_Acid_halides (yes) [ENDGRID,ENDFRAME]
```

Radio-buttons can be also arranged in a several rows/columns.

Example:

```
# { o_1_Option_11 [HORIZONTAL:2] o_2_Option_12 o_2_Option_13 o_2_Option_14 o_2_Option_1
# { o_1_Option_11 [VERTICAL:2] o_2_Option_12 o_2_Option_13 o_2_Option_14 o_2_Option_15
```

Creating a dialog with tabs.

Each page (tab) should be started with '%%%Page_Name' line. Then follows a page declaration.

Example:

```
MENU &File.Tabs Example
SYNT %%%Page1
SYNT # s_Text_1 ("aaa") [REQUIRED]
SYNT print $1
SYNT %%%Page2
SYNT # s_Text_2 ("bbb") [REQUIRED]
SYNT print $1
SYNT %%%Page3
SYNT # s_Text_3 ("ccc") [REQUIRED]
SYNT print $1
```

Creating a wizard dialog.

You may easily convert your tab dialog into wizard dialog by replacing '%%%' with '^'.

Example:

```
MENU &File.Wizard Example
SYNT ^^^Page1
SYNT # s_Text_1 ("aaa") [REQUIRED]
SYNT print $1
SYNT ^^^Page2
SYNT # s_Text_2 ("bbb") [REQUIRED]
SYNT print $1
SYNT ^^^Page3
SYNT # s_Text_3 ("ccc") [REQUIRED]
SYNT print $1
```

See also: `Askg`

Adding custom user functions to 'Insert Column'

This section contains description of the syntax to custom function for **Insert Column** dialog.

Example:

```
CFUN newgroup.Ligand_Strain
SYNT # (X)->r
SYNT funcLigStrain(%1)
```

Each column function start with **CFUN** which contains dot separated path in the form of folder1.folder2.funcname

Existing folders ID are:

- ♦ creat : "New"
- ♦ trans : "Transformations"
- ♦ arith : "Arithmetical"
- ♦ math : "Mathematical"
- ♦ search : "Find"
- ♦ str : "Text"
- ♦ chem : "Chemical"

For example to add new function into 'New' folder the path should look like: creat.MyNewFunc

The **SYNT** section should contain one or more **template description**. Each template description line start with '#'

Template describes the type of arguments and the result type.

The syntax:

```
(<arguments>)->(<result>), brackets are optional
arguments: space or comma-separated values in format:
        type:argname=default
Types combination of: i r s l w I R S X (Uppercase: array, lowcase: constant)
  "I" will accept both integer column (Iarray) and integer constant
  1I 1R 1S 1IR 1IRSX .. -- no constants allowed (only arrays)
```

Examples:

- ♦ (r r)->r # takes to real columns or two real values and return real column as a result
- ♦ (X r:curoff=0.5)->r # takes chemical and optional real argument with default 0.5

The line in the **SYNT** section which does not start with '#' will be treated as an arithmetic expression which calculates the column value. It may contain any arithmetic operators, constants, shell functions and user functions.

See `add column function` for details.

Commands

ICM commands have a certain structure:

- they use command words from this list: align assign build clear compare connect convert copy compress crypt delete display edit enumerate exit exclude filter find fix fork fprintf group gui keep learn load make menu minimize modify montecarlo move pause plot predict print printf quit query randomize read redo refresh rename restore rotate select set show sort split sprintf ssearch store strip superimpose test transform translate undisplay undo unfix unselect unix wait write
  alias antialias center color help history link list macro model sql web
  accessibility alignment amber angle area aselection atom axis background ball base bar bfactor bond born boundary box catalog cavity cell chain charge cmyk column command comment comp_matrix conf cpk csd cursor chemical cistrans database distance directory disulfide dot drestraint energy error evolution factor field filename font foreground frame function gamess genome gradient grid grob hbond header html hydrogen iarray icmdb idb image index info input integer intensity inverse iupac json kernel key label library logical loop limit margin material map matrix memory merit mol mol2 moldb molcart molecule movie molsar name nucleotide object occupancy oracle origin output page parray pattern peptide pipe pdb plane postscript potential preference preview problem profile project property prosite protein pharmacophore rarray reaction real regression reflection residue resolution ring rgb ribbon regexp rainbow sarray segment selection selftether separator sequence session site size skin slide salt solution stack stdin stdout stick sstructure string surface symmetry system svariable table term tether texture topology torsion trajectory transparent tree type unknown user variable vector version view virtual volume vrestraint vselection water weight window wire xstick
  add all append auto auxiliary binary bold bw cartesian clash chiral dash exact join fast fasta flat formal format full gcg gif global graphic heavy identity italic jpeg last left local mmcif mmff msf mute new none nosort number off on only pca pir pmf png pseudo pov reverse right similarity simple sln smiles smooth solid static stereo swiss targa tautomer underline unique wavefront xplor
- the they arguments consisting of constants, named shell variables or expressions including functions, e.g. `display ribbon Res(Sphere( a_H a_A 7.6 ))`
- the order of arguments of different data type is arbitrary
- at the end of the command one may have a list of additional shell variables that will be redefined temporarily only for the duration of this command, e.g. `montecarlo v_//x* mncalls=200 mncallsMC=20000 temperature=1000.`
- several commands in one line can be separated by a semicolon
- commands can return certain shell variables, like `i_out` , `i_2out`, `r_out` , `l_out` , `s_out` , `R_out` , `as_out` .. with useful output
- to suppress command output redefine those shell variables: `l_info=no` or `l_warn=no` (for some commands there is also a `mute` option)

---

add

A family of commands adding things. Some commands use `append` syntax instead It is also used as an option equivalent to `append` in `write` command.

add one or several columns or header elements to an existing table

Adding a single column/header you may add a column (or columns) to an existing table *T* or create a new one if the specified name does not exist.

`add column` *T I*|*R*|*S*|*P*|*i*|*r*|*s* [`name=` *s* [`append`|`delete`]] [`index=` *i_pos*] [`mute`] [`local`]

`add header` *T I*|*R*|*S*|*P*|*i*|*r*|*s*|*M*|*etc* [`name=` *s* [`append`|`delete`]] [`mute`] [`local`]

(to add a row see `add` *`table args`*)

Adding multiple column/headers

`add column` *T I|R|S|P|i|r|s I|R|S|P|i|r|s* .. [`name=` *S* [`append|delete`]] [`index=` *i_pos*] [`mute`] [`local`]

`add header` *T I|R|S|P|i|r|s|M|etc I|R|S|P|i|r|s|M|etc* .. [`name=` *S* ..] [`index=` *i_pos*] [`mute`] [`local`]

this command adds one or several columns to an existing table **in** the *i_pos* column (in other words if you want you column to be in the 2nd position, specify 2 as an argument). The columns are append to the end of the table by default. If the table does not exist the command will create a new table.

It an integer, string, or real are specified as an argument instead of a column-array, this value is multipled to create a column of the appropriate size.

Options:

- ♦ `append`: if the `name` option is specified prevents overwriting the column with the same name, instead modifies the provided name (e.g. 'A' -> 'A_1')
- ♦ `delete`: if the `name` option is specified overwrites the column with the same name. Without `delete` (default) it will be overwritten only if the data type is the same.
- ♦ `mute` : The `mute` option suppresses the info (equivalent to temporarily setting `l_info=no`).
- ♦ `local` : for use in macros ans shell functions: allows for local tables independent from tables with the same name at higher levels.

Examples:

```
add column t {1 2 3} # create a new table
add header t "A new table" name="title" # add a string to the header section
add column t {"a","b","c"} name="AA" # column AA is appended
add column t {"x","y","z"} index = 2
# adding a chemical array
add column t Parray({"CC","CC(O)=O","CCO"} smiles) name = "mol" index = 1

# adding multiple arrays
add column t {1 2 3} {3 2 1} {"a","b","c"} Parray({"CC","CC(O)=O","CCO"} smiles) name={
t
 #>T t
 #>-A-----------B-----------C-----------mol--------
    1           3           a           "CC"
    2           2           b           "CC(O)=O"
    3           1           c           "CCO"
```

The columns can also be `functions`, e.g.

```
add column t {1. 2. 3.} {2 3 4}
add column t function="A+B" name="AplusB"
add column t function="Log(A,10)" name="log10A"
```

Use `add column` inside a macro:

```
macro ResAreas  rs_sel
  rs_sel = Res( rs_sel & a_*.!W )
  show surface area Mol(rs_sel) Mol(rs_sel)
  add column t Name(Res(rs_sel) full) Area(Res(rs_sel)) Area( a_1.A/A )/Area( a_1.A/A t
  set format t.sel "<!--icmscript name=\"1\"\ndisplay xstick %1\ncolor xstick cpk %1 &
  if(Type(resAreas)!="unknown") delete resAreas
  rename t "resAreas"
  keep resAreas
endmacro
```

See also: `move column` , `add column function`

Add dependent columns

`add column` *T* `function=`*s_expression* [`name=`*s*] [`index=` *i_pos*]

Adds a column defined by the *s_expression*, which may contain operations with other columns in the same table. The generating expression information is attached to the column, which allows one to recalculate the values in the column using the same expression. The following functions are supported:

Basic arithmetical operations on columns are supported, examples:

```
add column t {1. 2. 3.} {1. 2. 3.}
add column t function="A+B" name="C"
add column t function="A-B" name="D"
add column t function="A*B" name="E"
add column t function="A/B" name="F"
add column t function="A**0.5 + B**0.5" name="G"
build column t
show t
```

The following mathematical and data conversion functions are supported:

```
 Ceil, Floor, Log,  Sqrt, Sign, String, Power
```

Examples:

```
add column t {1. 2. 3.} {1. 2. 3.}
add column t function="Sqrt(A)" name="C"
```

Chemical functions Examples of chemical functions

```
add column t function="Nof_Atoms(mol,'*')" name="nof1"  # all atoms
add column t function="Nof_Atoms(mol,'[!H]')" name="nof2"
add column t function="Nof_Atoms(mol,'[C,O]')" name="nof3"
add column t function="Nof_Atoms(mol,'[H]')" name="nof4"
add column t function="MolWeight(mol)" name="molWeight"
add column t function="MolLogP(mol)"   name="molLogP"
add column t function="MolLogS(mol)"   name="molLogS"
add column t function="MolPSA(mol)"    name="molPSA"
add column t function="MolVolume(mol)" name="molVolume"
add column t function="MoldHf(mol)"    name="moldHf"
```

User-defined functions

A user can defined custom function to be used in column formula expression

Example:

```
function ligStrain( P_chem )  # returns strain for a given 3D chemical
  vwMethod = "exact"
  dielConst = 2.
  read mol P_chem name="LIGSTRAIN"
  build hydrogen
  set type charge mmff
  convert auto
  minimize cartesian "mmff" mncalls=1
  newE = Energy("ener")
  minimize cartesian "mmff" 5000
  baseE = Energy("ener")
  r_ligandStrain = newE - baseE
  delete a_
  return r_ligandStrain
endfunction

# assumes that t_3D exists and contains 3D chemicals in the mol column
add column t_3D function="ligStrain(mol)" name="strain"   # strain for every row in the
```

ICM built-in function

ICM build-in functions can also be used in the expression with "Icm::" prefix.

Example:

```
add column drugs function = "Icm::Sum(Icm::Unique(Icm::Sort(Icm::Sarray(A.dosages.dosag
```

Recalculating. To recalculate use the `build column` *column_name* command.

Examples:

```
add column T Chemical( {"c1(c(nc(N)nc1O)O)N", "c1c[nH]c2C(N=C(N)Nc12)=O"} ) name="mol"
add column T function="MolWeight(mol)" name="MW"
add T # add new row
```

```
T.mol[3] = Chemical("CC")
#
build column T.MW # recalculate mol. weights, setting the value for the new row

add column T2 {1 2 3} {4 5 6}
add column T2 function= "A+B"  # sum of two columns
```

See also: `build column` to update values, `Parray` (*X* [*s_func*]) to add a fixed column

Adding real arrays as matrix rows

`add matrix` [*M*] *R*|*M2*

adds a matching row *R* or a matrix with the matching number of columns to matrix *M*, by stacking extra rows at the bottom. If the matrix does not exist it is created with the default name (the name is returned in `s_out`) Example:

```
add matrix M {1. 2.} # creates new matrix M
add matrix M {3. 3.} # adds a row
show M
 #>M M
 1. 2.
 3. 3.
add matrix M Matrix(2) # adds two rows
```

Note that to extend the matrix horizontally (adding columns) can be done with the double-slash operator (*M1* // *M2*).

add slide to a slideshow.

`add slide` [*i_posInCurrentSlideshow*] [*s_slideTitle*] [`comment` = *s_slideComment*] [`display=` "*-option*|*-option*"]

adds a slide to the `slideshow` table. This table contains one `parray`, called `slideshow.slides` . If the slide position is not specified the slide will be added to the end. Alternatively, it will be inserted *after* the specified *i_posInCurrentSlideshow*

Normally the slide is saved with window layout, and graphical parameters. Those can be ignored if you add the `display="-.."` option. The options can be used either in the **on** mode, e.g. "-layout", or in the **off** mode, e.g. "+layout" (all **on** by default) :

- `"-layout"` # ignores the window/panel layout
- `"-smooth"` # ignores smooth view transitions between slides
- `"-add"` # do not overwrite the previous slide views, just add to it
- `"-gf"` # ignore graphical representations, inherit them
- `"-color"` # ignore colors , inherit them
- `"-labeloffs"` # do not display labels
- `"-viewpoint"` # ignore viewpoint changes
- `"-graphopt"` #
- `"-mol"` # do not display the chem-table window
- `"-grob"` # do not display grobs
- `"-map"` # do not display maps
- `"-all"` # switches off all the above properties

Example:

```
build string "ala ala ala"
display ribbon a_
display xstick a_/12,13 magenta
add slide "My View" comment = "Two magenta residues" display="-layout"
undisplay # hide all
# wait..
display slide "My View" # bring it back
```

See also: `display slide` , `Slide`

Add / insert table rows. Append tables.

`add` *T_1* [*i_RowNumber*] [*T_2* | `row_selection` | `number=`*i_nofRows*] [`simple`]
add/insert rows (or another table with the same coloumns) to table *T_1* at the target row position *i_RowNumber* . Use 1 (one) if you need to insert the first line. If the second table or selection is not provided, the command adds an empty row. In this case you can add *number* option to specify the number of rows to add/insert. The *row_selection* can contain rows from the *same* table or from a *different* table with a matching column structure. In the latter case, the columns may be matched by their names regardless of column order. Default values are inserted for all absent columns. The defaults for an empty line are empty string or zero value for strings or numbers, respectively. The target position will then correspond to the index of the first inserted row.

simple option toggle column matching order 'by position' instead of default 'by name'.

From version 3.6-1e the `add` *tableName* command also returns the current row as `i_out` .

Examples:

```
group table t {1 2 3} "a" {"b","d","e"} "b"
show t
 #>T t
 #>-a-----------b----------
    1           b
    2           d
    3           e

add t 1 # insert empty line before 1st
show t
 #>T t
 #>-a-----------b----------
    0           ""
    1           b
    2           d
    3           e

group table t {1 2 3} "a" {"b","d","e"} "b" # recreate the table
add t 3 t[1]  # insert a duplicate of 1st row after the 2nd
show t
 #>T t
 #>-a-----------b----------
    1           b
    2           d
    1           b
    3           e

group table t {1 2 3}  "a"  {"b","d","e"} "b"  # recreate the table
group table tt {1 2 3} "c" {"b","d","e"} "b" {4 5 6} "a" # another table
# order is diffferent, extra column present
add t 3 tt[1:2]  # or add t 3 tt.aa<3
show t
 #>T t
 #>-a-----------b----------
    1           b
    2           d
    4           b
    5           d
    3           e
```

alias

`alias abbreviation` *word1 word2 ...*
create alias
`alias delete abbreviation`
delete alias
It is important that the abbreviation is not used in the ICM-shell. The same names can not be given later to ICM-shell objects.
Alias may contain arguments $0, $1, $2, etc. ICM-shell will pick space-separated words following the alias name and substitute $1, $2, etc. arguments by the specified argument. $0 stands for all the arguments after the alias name.
Examples:

```
 alias seq sequence                     # seq will invoke sequence
```

```
alias delete seq                       # delete alias name seq
alias dsb display a_//ca,c,n           # abbreviate several words to
                                       # reduce typing efforts
                                       # aliases with arguments
alias NORM ($1-Mean($1))/Rmsd($1)
show NORM {6,7,8,4,6,5,6,7,5,6}        # make sure there is no space
```

align

align number: renumber residues sequentially

`align number` *rs_residuesToBeRenumbered i_first|s|I|S* [`molecule`]
`align number` *ms_chainsToBeRenumbered* [*i_firstNumber*]
renumber selected residues, or residues in selected molecules or objects sequentially in all of them from starting one or the specified first number. May be useful to deal with messy numbering in some pdb-files. Option `molecule` will start numbers from 1 or *i_first* in *each* molecule. Chain ids are also allowed, e.g. `set number a_/13 "13A"`. Multiple residues can be set with integer or string arrays of labels. If integer array contains the same numbers, e.g. `10,10,10` the labels will get the insertion characters, e.g. `10`,10A,10B .

Examples:

```
read pdb "1crn"
align number a_1             # renumber all res. 1 to N
align number a_1/10:20 101   # just the selected residues from 101
align number a_1       101   # renumber all res. 101 to 100+N
read pdb "2ins"
align number a_/* 1
align number a_/* 1 molecule # each chain starts from 1
```

`align number` *ms_chainsToBeRenumbered seq_master* [*i_offset*] renumber the residues of the selected molecule according to *seq_master* master sequence which is aligned to the sequence of the selected chain. The alignment (pairwise or multiple) need to be `linked` to the molecule/chain and both the chain sequence and the master sequence need to be covered by the alignment. The molecular sequence can be generated with the `make sequence` [*ms_chainsToBeRenumbered*] command.
This command may be useful in cases in which a structural model does not represent the entire sequence because of omitted loops, N- and C- termini, while you still want to keep the numbering according to the full master sequence. You might want to use the command also on models by homology generated with the `build model` command.
Example:

```
seqmaster = Sequence("ACDEFGHIKLMNPQRST")
build string   "--DEFGH-----PQRST"  # dashes are skipped
make sequence a_1 name="seqmodel"   # sequence is auto-linked
a = Align(seqmodel,seqmaster)       # linked alignment
align number a_1 seqmaster
# Info> residues of a_def.m renumbered by sequence 'seqmaster' from alignment 'a'
display residue label
```

align: ICM multiple alignment algorithm

`align` *ali_SequenceGroupName* [`tree=`|`filename=` *s_epsFileName*]

`align sequence` [`selection`] | *seq1 seq2* .. | *seedSequence* [`min_seqID` (`20.`)] [`name=`*s*]
make a multiple alignment of specified sequences. The sequence group may result from the `group sequence` *s_groupName* command. The input arguments include the following:

- *seq1 seq2 ...* : explicitly specified
- `sequence` : all sequences in the shell
- `sequence selection` : all sequences selected in GUI
- *seqGroup* : sequences grouped together previously
- *seedSequence* : sequences in the shell similar to the specified with optional *min_seqID* (default 20%).

For pairwise alignment use the `Align(` *seq1* seq2 `)` function. The algorithm includes the following steps (inspired by corridor discussions with `Des Higgins, Toby Gibson and Julie Thompson`):

1. align **all** sequence pairs with the ICM `ZEGA` algorithm, and calculate pairwise `distances` between each pair of aligned sequence with the Dayhoff formula, e.g. the distance between two identical sequences will be 0. , while the distance between two 30% different sequences will be around 0.5. The distance goes to an arbitrary number of 10. for completely unrelated sequences. The distance matrix $\boldsymbol{D}_{ij}$ can later be extracted from the alignment with the `Distance(` *ali_* `)` function.
2. build an evolutionary tree from $\boldsymbol{D}_{ij}$ with the "neighbor-joining algorithm" of `Saitou, N., Nei, M. (1987)` to determine the order of the alignment and calculate relative weights of sequences and profiles from the branch lengths. The tree will be saved in the file defined by the `tree=` *s* or `filename=` *s* option . Starting from version 3.5-2 the `aligTree.eps` file is NO LONGER saved by default). The so-called `Newick tree` description `string` will be saved in `s_out` .
3. traverse the tree from top to bottom, aligning the closest sequences, sequence and profile or two profiles. After each Needleman and Wunsch alignment, build the profile.
4. generate the final neighbor-joining evolutionary tree and write the PostScript file with the tree to disk.

Examples:

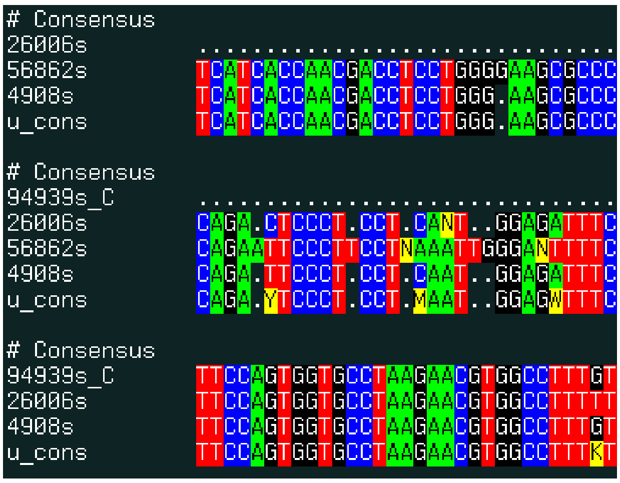
```
   read sequences s_icmhome+"zincFing"
   list sequences                # see them, then ...
   group sequence alZnFing       # group them, then ...
   align alZnFing                # align them
   align alZnFing filename="znTree.eps"  # eps file with a tree

  read sequence swiss web "12S1_ARATH"
  read sequence swiss web "12S2_ARATH"
  group sequence arath
  align arath
```

EST,DNA alignment and assembly

`align` new *ali_sequenceGroup* [*seq_seed*]
multiple alignment of ESTs and genomic DNA and consensus derivation. This command uses the external the sim4 program to generate pairwise alignments between expressed DNA sequence and a genomic sequence. The program can be downloaded from the `http://globin.cse.psu.edu/globin/html/docs/sim4.html` site.

The procedure has the following steps:

- sequences are sorted by length
- the longest sequence is chosen as the seed sequence unless it is explicitly provided
- the longest sequence from the remaining set is aligned to the seed sequence using the external `sim4` program.
- the output of this program is parsed and translated into the icm alignment
- the **consensus** sequence is created and becomes the master sequence
- the procedure is repeated until all the sequences are processed
- the multiple sequence alignment is further cleaned to compress spurious gaps when possible. This cleaning makes the consensus much more compact.



```
# Consensus
26006s          ...............................
56862s          TCATCACCAACGACCTCCTGGGGAAGCGCCC
4908s           TCATCACCAACGACCTCCTGGG.AAGCGCCC
u_cons          TCATCACCAACGACCTCCTGGG.AAGCGCCC

# Consensus
94939s_C        ...............................
26006s          CAGA.CTCCCT.CCT.CANT..GGAGATTTC
56862s          CAGAATTCCCTTCCTNAAATTGGGANTTTTC
4908s           CAGA.TTCCCT.CCT.CAAT..GGAGATTTC
u_cons          CAGA.YTCCCT.CCT.MAAT..GGAGWTTTC

# Consensus
94939s_C        TTCCAGTGGTGCCTAAGAACGTGGCCTTTGT
26006s          TTCCAGTGGTGCCTAAGAACGTGGCCTTTTT
4908s           TTCCAGTGGTGCCTAAGAACGTGGCCTTTGT
u_cons          TTCCAGTGGTGCCTAAGAACGTGGCCTTTKT
```

The result of this command is best

displayed with the `show color` *`ali_`* command.

An example:

```
read sequence "http://www.ncbi.nlm.nih.gov/UniGene/" + \
    "download.cgi?ID=5198&ORG=HsLINE=1" #
read sequence "../Hs5198"
group sequence unique u # squeeze out obvious redundancies and form group 'u'
align new u             # form multiple alignment and build consensus
show color u
```

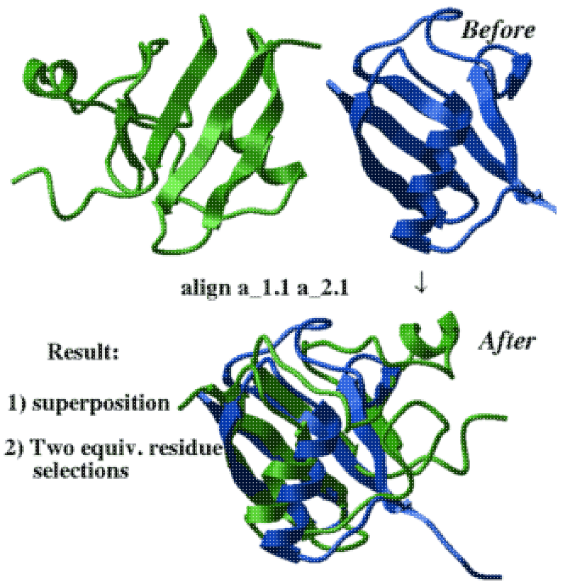
See also:

- ♦ `filtering, group sequence unique=".."`
- ♦ `Trans` ()
- ♦ show [color] *ali_*

---

align two molecules by their backbone topology

`align` [`distance`] *ms_1 ms_2* [*i_windowSize* (`15`)] [*r_seqWeight* (`0.5`)]



This command finds the residue *alignment* (or residue-to-residue correspondence) for two arbitrary molecules having superposable parts of the backbone conformations. The structural alignment identification and optimal superposition is primarily based on the C-alpha-atom coordinates, but the sequence information can be added with a certain weight (the default value of *r_seqWeight* is 0.5 which was found optimal on a benchmark). The structural alignment algorithm is based on the ZEGA (zero-end-gap-alignment) dynamic programming procedure in which substitution scores for each i,j-pair of residues contain two terms:

- ♦ structural similarity in a *i_windowSize* window between two fragments surrounding residues i and j, respectively. This similarity is calculated as local `Rmsd` of the residue label atoms (these atoms are C-alpha atoms by default but can be reset to other atoms with the `set label` command, e.g. `set label a_*.//cb`). If the option `distance` is specified the deviation of the interatomic distances between equivalent *pairs* of atoms (so called *distance rmsd*) is calculated instead of a more traditional root-mean square deviation between atom coordinates of equivalent atoms. The latter method is less accurate but an order of magnitude faster.
- ♦ sequence similarity (if *r_seqWeight* > 0.). Average local sequence alignment score in the *i_windowSize* window is calculated for i,j-centered pair of fragments. In this sense this sequence similarity is different from the one used in pure sequence alignment (see the `Align` function), in which just the i,j residue pair is evaluated. The default value of *r_seqWeight* of 0.5 is rather mild (about a half of the structural signal).

The output:

- ♦ `ali_out` contains structural alignment (if sequences linked to the molecules do not exist, they will be created on the fly). The alignment can be further edited with the interactive alignment `alignment editor`.
- ♦ `as_out` contains the `residue selection` of the aligned residues in the first molecule

- ♦ `as2_out` contains the `residue selection` of the aligned residues in the second molecule
- ♦ `M_out` , the matrix of local structural/sequence similarity in a window is retained and can be visualized by:
- ♦ `r_2out` the result RMSD

Example:

```
read pdb "1ql6"
read pdb "2phk"
align a_1ql6. a_2phk.
make grob color 10.*M_out name="g_mat # x,y,z scales
display g_mat
# or
plot area M_out display grid link
```

See also:

- ♦ `Align(` *seq_1 seq_2* `distance|superimpose`). This function creates the first unrefined structural alignment as described above.
- ♦ `find alignment` which refines initial structural alignment.

The overall result of the `align` command is equivalent to:

```
  a = Align(... superimpose )  # superposition/RMSD based local str. alignment
  a = Align(... distance    )  # distance RMSD based local str. alignment

  find a superimpose 4.0 0.5
```

Example:

```
  read pdb "1brl"
  read pdb "1nfp"
  rm a_*.!A
  display a_*.//ca,c,n
  color molecule a_*.
  align a_2.1 a_1.1
  center
  show String(as_out) String(as2_out)
  color red as_out
  color blue as2_out
  show ali_out
```

align heavy command for multiple alternative structural alignments.

`align heavy` *rs_1 rs_2* [*r_rmsd*] [*i_windowSize* [*i_minFragment*]] [*r_elongationWeight*]
This method, as opposed to the default `align ms_1 ms_2` generates many possible solutions and does not depend on sequential order of the secondary structure elements. However, it leads to a combinatorial explosion and is intrinsically less stable computationally, and generally requires more time. The command finds the optimal 3D superposition between two arbitrary molecules/fragments (two residue selections *rs_1* and *rs_2*).
The procedure generates structural fragments of certain initial length and superimposes all of them to calculate the structural similarity distance. Then the "islands" of similarity are merged into larger pieces. This process is controlled by the following arguments: *i_windowSize* is the residue length of structural fragments for the initial fragment superposition. Fragment pairs with the rms deviation less than *r_rmsd* are then combined, giving composite solutions of total residue length larger than *i_minFragment.* Acceptance or rejection of the composite solutions is governed by the following score (the smaller, the better)
score = rmsd - (1.37 + Sqrt(1.16 * length - 15.1)), length >= 14
If ***length > 14*** , we use linear extrapolation of the score dependence:
score = rmsd - (1.37 + 1.068*(length-13))
The score is required to be less than *r_rmsd.* Practically, for longer fragments one can find much larger RMS deviations according to the length correction of the score.
Defaults:

- ♦ *r_rmsd* = 1. A
- ♦ *i_windowSize* = 15 residues
- ♦ *i_minFragment* = *i_windowSize*
- ♦ *r_elongationWeight=0.1*

There may be several different reasonable solutions. All the solutions are sorted, shown and stored in the memory. The two output selections `as_out` and `as2_out` contain the best scoring solution. Any solution can be `load`ed and displayed. Additionally, a residue alignment is created for each solution. The decision about which residues are aligned is based on the overall score described above for the of combined fragments.
See also: `How to optimally superimpose without the residue alignment`
Example:

```
 read pdb "4fxc"
 read pdb "1ubq"
 display a_*.//ca,c,n
 color molecule a_*.
 align heavy a_1.1 a_2.1 12 1.5 .1
 center
 load solution 2            # load the second best solution
 color red  as_out
 color blue as2_out
 for i=1,10
   load solution i
   color molecule a_*.
   color red  as_out
   color blue as2_out
   pause                     # rotate and hit 'return'

 endfor
```

Note. Increase *i_minFragment* parameter (12 in the above example) to something like 20 if the program hangs for too long. Interrupt execution with the ICM-interrupt (**Ctrl **) if you want only the top solutions.

append (commands)

There is a family of commands starting with the `append keyword`. They are usually used to add sub-elements to a compound object like an `alignment` or a `stack`. In many cases ICM uses `add` syntax instead of `append`.

Appending sequences to a sequence group or an alignment

`append` *ali_seqGroup seq_1 seq_2 .. .*
appends sequences to a sequence group. This may be required if you formed a sequence group for future alignment or filtering/compression and you want to append additional sequences to it.

Examples:

```
read sequence group "bunch.seq" name="xx" # group xx is formed
append xx my_seq  # appending your sequence to xx
group xx unique   # filter out identical ones
align xx

read sequence swiss web "12S1_ARATH"
read sequence swiss web "12S2_ARATH"
group sequence name="arath"
read sequence swiss web "14310_ARATH"
append arath 14310_ARATH
align arath
```

Appending a molecule or a ligand stack to an existing stack

`append stack` *s_ligandStackFileName* [*i_maxConf*]

`append stack` *os_ligandObject*

this command takes a stack which corresponds to a receptor object and appends each `conformation` in the stack with a conformation of the ligand. If the ligand conformation can be taken from either from a stack file, this command will combine each `conf` from the main stack with **all** conformations from the file. The *i_maxConf* argument will set the limit on how many conformations are taken from the ligand stack (i.e. `append stack "lig.cnf" 1` will combine only the first conformation of the ligand)

If the second argument is an ICM object, each conf of the current stack will be extended with the variables from the ligand. Now the ligand object can be appended to the receptor object with the `move` command and the new combined object can use the expanded stack.

```
build string "ACDEF"  # the "ligand" peptide
rename a_ "Lig"
translate a_ {10. 0. 0.} # shift not to overlap with a_Rec.
montecarlo v_//!?vt*     # created Lig.cnf stack

build string "RSTVW"  # the "receptor" peptide
rename a_ "Rec"
montecarlo v_//!?vt*     # created Rec.cnf stack

move a_1. a_2.     # ligand must be the 1st argument
append stack "Lig.cnf" 4 # combine up to 4 best ligand confs
minimize stack     # minimizes each stack conf
load conf 1        # check them out
```

append two tables via two columns with matching values

`append` *t1.A t2.B*
Append rows of table *t2* to table *t1* by rows corresponding to unique column *t2.B* . The *t1.A* column values do not need to be unique.

```
group table people {"J","C","M"} "p" {"MS","MS","MS"} "orgid"
group table orgs  {"MS"} "id" {"Molsoft"} "name"
append people.orgid orgs.id
people
 #>T people
 #>-p-----------orgid-------name-------
    J           MS          Molsoft
    C           MS          Molsoft
    M           MS          Molsoft
```

This command is a particular case of a more general `join` command.

See also the `add table` command for adding rows from a column with identical column structure (e.g. `add t tt`).

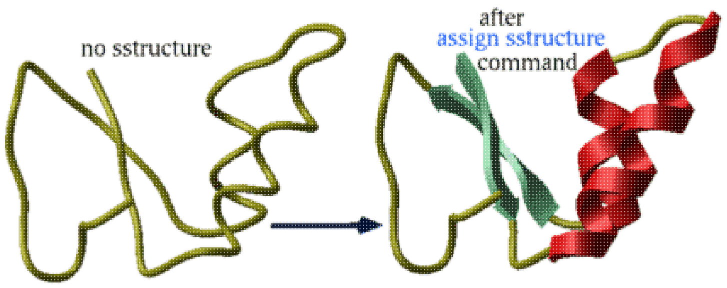
assign

assign sstructure: derive secondary structure from a pattern of hydrogen bonds

`assign sstructure` *rs* [{ *s_SecondaryStructTypeCharacter* | *s_SSstring* }]
Manual assignment of a desired secondary structure annotation to a residue fragment

`assign sstructure` *rs* { *s_SecondaryStructTypeCharacter* | *s_SSstring* }
assign specified secondary structure to the selected residues `rs_` , e.g.

```
 read pdb "1crn"
 assign sstructure a_/* "_"  # make everything look like a coil
 cool a_
 assign sstructure a_/1:10 "HHHH_EEEEE"
 cool a_
```



This command does not change the geometry of the model, it only formally assigns secondary structure symbols to residues. **Note:** to **change the conformation** of the selected residue fragment, according to a desired secondary string, use the ICM -object and the `set` command applied to both sequences and molecular objects.

Automated derivation and assignment of secondary structure from atomic coordinates
`assign sstructure` *rs*
If the secondary structure string is not specified, apply ICM modification of the DSSP algorithm of automatic secondary structure assignment `(Kabsch and Sander, 1983)` based on the observed pattern of hydrogen bonds in a three dimensional structure.
The DSSP algorithm in its original form overassigns the helical regions. For example, in the structure of T4 lysozyme (PDB code **103l**) DSSP assigns to one helix the whole region a_/93:112 which actually consists of two helices a_/93:105 and a_/108:112 forming a sharp angle of 64 degrees. ICM employs a modified algorithm which patches the above problem of the original DSSP algorithm. Assigned secondary structure types are the following: "H" - alpha helix, "G" - 3/10 helix, "I" - pi helix, "E" - beta strand, "B" - beta-bridge, "_" or "C" - coil.
Examples:

```
 nice "1est"       # notice that many loops look like beta-strands
 assign sstructure # now the problem is fixed
 cool a_
```

See the `set` *`rs_ s_SecStructPattern`* command to actually set new phi, psi angles to a peptide backbone according to the string of secondary structure.

assign sstructure segment

`assign sstructure segment` [*ms_molecules*] # *ms_ICMmoleculesPreferable*
create simplified description of protein topology (referred to as `segment` representation). Segments shorter than `segMinLength` are ignored. The `current object` is the default. This command will work both on un `convert`ed pdb files as well as the pdb files. However the resulting secondary structure will be BETTER when the structure is converted and hydrogens are added.
See also `show segment`, `ribbonStyle`, `display ribbon`. `convert` `convertObject`

break

is one of the `ICM flow control` statements. It permits a `loop` (e.g. `for` or `while`) to be `broken` before calculations have completed.
Examples:

```
   for i = 1, 8
     print "Now i = ", i, "and it goes up"
     if (i == 4) then
       print "... but at i=4 it breaks, Ouch!"
       break
     endif
  endfor
```

See also `goto` .

assign residue

Assigns residue structure to a peptide or a protein. Sometimes when you read a peptide or protein from MOL or MOL2 with no residue information present it is treated as a single residue small molecule. This command allows to restore residue layer.

`assign residue` *os1*

Example:

```
build string "EACARVAAACEAAARQ"
read mol Chemical( a_ exact hydrogen )  name="xxx"  # read it as a single residue small
assign residue a_   # restore residue structure
```

```
Sequence( a_1. )
Sequence( a_2. )
```

build

The build family of functions allows one to create molecular objects

- from sequence file (`build s_seqfile`)
- from sequence string (`build string`)
- from a linear chemical notation (`build smiles`)
- from a sequence and a template *by homology* (`build model`)

It also adds implied hydrogens (`build hydrogen`) to a molecule and to find a loop in a database (`build loop`)

build one atom and rebuild hydrogens

`build atom` *as1* `[simple]` `[s_elementName=("c")]` `[i_bondType=(1)]`

`build pseudo` *as_inICMobj*
by default it will add a carbon to the selected atom in a non-ICM object and rebuild hydrogens for the affected atoms. Use the `strip` command for ICM objects.

Options and arguments:

- `simple` does not rebuild hydrogens.
- *s_elementName* is a string with the name of the chemical element.
- *i_bondType* is 1 for a single bond (the default), 2 for a double bond and 4 for a triple bond.

Example:

```
build smiles "CC(C)Cc1ccc(cc1)C(C)C([O-])=O" name="ibuprofen"
strip a_ibuprofen.
build atom a_ibuprofen.m//c1 "n"
build atom a_ibuprofen.m//c1 2
```

See also:

- `make bond`
- `delete bond`
- `delete atom`

Recalculating dependent columns

`build column` *T.col|T*

Rebuilds all values in a dependent column *T.col*

`build column` *T* | *T_row_selection*

Rebuilds all dependent columns in the table *T_* or row selection (e.g. `T[12]`, or `T.ID==123`) If column A depends on column B and column B depends on other columns, column B will be calculated before column A.

Examples:

```
add column T {2 5 1} name="B"
add column T function="A + B" name="C"
add column T function="C + B" index=1 name="D"
T.A[1] = 10
build column T[1] # should change values of C and D in the first row
```

See also: `add column function`

Building object from sequence file

`build` *s_IcmSeqFileName* [`library=` { *s_libFile* | *S_libFiles*}] [`delete`]
reads *s_IcmSeqFileName.se* `ICM-sequence file` and builds an ICM molecular object. This sequence file is different from a simple sequence file and contains three (sometimes four) character residue names defined in the `icm.res` residue library file (try `show residue types` to see the list).

Use command `build string` if you want to build an object from a string with one letter coded sequences or a named sequence. E.g. `build string "ASDGF"` or "ASD;DERR" or "nh2 ala his cooh"

To get a D-amino acid instead of L-ones simply use D as a prefix: `Dala Darg.` Specify N- or C-terminal modifiers directly in the file if needed. The build command will create them in some default conformation (extended backbone with different molecules oriented around the origin as a bunch of flowers). Several molecules can be specified in the ICM sequence file.
Residue names may contain numbers (i.e. 4me). However, the residue numbers with a modification character, such as 44a, 44b should contain a slash before the modification character (i.e. 44/a , 44/b). An example in which we create a sequence of residues ala and 4me with numbers 2a and 2b, respectively: "se 2a ala 2b 4me".
The *library* option lets to temporarily switch the library file. The same result may also be achieved by redefining the `LIBRARY.res` array of the `LIBRARY` table.
The *delete* option temporarily sets the `l_confirm` flag to `no` and the old object with the same name gets overwritten. Examples:

```
 build "def"  # def.se file
 build s_icmhome + "alpha.se"  # alpha.se file
 build "wierd" library="mod.res"  # get residues from mod.res
#
 LIBRARY.res = {"icm","./myres"}
 build "s"
```

build tautomer

`set tautomer` *ms*|*rs*

prepares internal data for quick switching between different tautomer states of small molecules *ms* or histidine *rs_his* by relative tautomer number or histidine tautomer name.

You need to call this command if you plan to sample different tautomers in `montecarlo` command (~~tautomer option)

Example:

```
build string "AHW"
build tautomer a_/his # adds a hydrogen and hydrogen masks to allow the switching
monecarlo reverse tautomer
```

See also: `set tautomer`

Building model by homology

`build model` *seq_1 seq_2 ... ms_Templates ...* [*ali_1 ...*] [`margin=` { *i_maxLoopLength*, *i_maxNterm*, *i_maxCterm*, *i_expandGaps* }
build a comparative model (homology model) of the input sequences based on the similarity to the given molecular objects. The margin arguments:

name	default	description
i_maxLoopLength	999	longer loops are dropped
i_maxNterm	1	the maximal length of the N-terminal model sequence which extends beyond the template
i_maxCterm	1	the maximal length of the C-terminal model sequence which extends beyond the template
i_expandGaps	1	additional widening of the gaps in the alignment. End gaps are not expanded

Possible modes:

- simple one-to-one mode: build model seq_1 [ms_1] [ali_1]
- N sequences - N corresponding molecules: `build model` *seq_1 seq_2 .. seq_N ms_1,2,..N* This mode requires the `minimize tether` command to complete the construction.

Examples:

```
 l_autoLink = yes
 read pdb "x"
 read alignment "sx"
 build model 1y6 a_
 ribbonColorStyle = "alignment" # grey-gaps, magenta-insertions
 display ribbon

 read pdb "2ins"  # multichain
 a = Sequence( "GIVEQCCASV CSLYQLENYC N" )
 b = Sequence( "VNQHLCGSHL VEALYLVCGE RGFFYTPKA" )
 c = a
 d = b
 build model  a b c d a_1.
 minimize v_//V "tz" 1000
# or   minimize tether
# Now optimize the side chains
 selectMinGrad = 1.5
 set vrestraint a_/*
 montecarlo fast v_/!I/x*
# !It means residues which are not Identical to their template residues
# use  refineModel to energetically optimize the model
```

The algorithm performs the following steps:

Alignment adjustment: modifies the alignment according to *i_expandGaps*, and prepare a sequence with the ends and the long loops truncated according to the alignment and the { *i_maxLoopLength* , *i_maxNterm* , *i_maxCterm* } parameters.
Building a straight polypeptide from the model sequence: builds a full-atom polypeptide chain for this new sequence. The residues in your model are numbered according to the template and all the inserted loops residues are indexed with 'a','b', etc. E.g. the numbering may look like this: `200`,201,203,204,204a,204b,204c,205 ... This numbering allows one to follow more easily the correspondence between the template and the model. If you do not like this numbering scheme, just use the

```
align number a_/*
```

command and the model residues will be renumbered from 1 to the number of residues.
Backbone topology transfer: inherits the backbone conformation from the aligned (but not necessarily identical) parts of the known template
Identical side-chain building: inherits conformations of sidechains identical to their template in the alignment
Non-identical side-chain placement: assigns the most likely rotamer to the side chains not identical in alignment. If you want to do more than that apply:

```
set vrestraint a_/*  # assigns the rotamer probabilities
montecarlo fast v_/Cx/x* # x* selects for all chi (xi) angles
```

You can also manually re-optimize any side chains either interactively (right-mouse click on a residue atom, then select Shake Amino-Acid Side-Chain) or from a script, e.g. for residue 14:

```
set vrestraint a_/*  # assigns the rotamer probabilities
montecarlo v_/14/x*
ssearch v_/14/x*     # systematic conformational search for the 14-th sidechain
```

Loop searches:

searches the `icm.lps` which may contain entire PDB-database for suitable loops with matching loop ends and as close loop sequence as possible, inserts them into the model and modifies the side-chains according to the model sequence.
The loop file can be easily customized, updated and rebuilt with the `write model [append]` command in a loop over protein structures. To use your custom loop file, redefine the

`LIBRARY.lps` variable.
Loop refinement and storing alternatives: adjusts the best loops found and keeps a stack of loop alternatives which can later be tested (see the Homology gui-menu).

The output

The build model command returns the following variables:
`LoopTable` master table containing list of all the loops, their conformation in alphanumeric code, a measure of the deviation of the database loop ends and the model attachment sites, the loop length and the numerical conformation type (not really important). E.g.

```
#>T LoopTable
#>-1_Loop------2_Conf------3_Rmsd------4_Nof-------5_Type-----
   a_ly6.a/7:10 31R21       0.1         11          1
   a_ly6.a/60:63 1RRR32      0.1         8           1
   a_ly6.a/43:46 211331RRRR  0.240658    4           1
```

Individual loop tables
Tables called `LOOP1` , `LOOP2` , etc. for each inserted loop. The tables contain the coded conformational string, relative energy, the position of the offset in the structure database file (`offset`) to be able to extract this loop again, and the `rmsd` of the loop ends. Example:

```
icm/ly6> LOOP1
#>T LOOP1
#>-Conf--------energy------offset------rmsd-------
   31R21       0.          3623594     0.092104
   31RR2       1.519275    3427772     0.083372
   R1121       1.612712    3750108     0.097777
   R1R32       1.639177    1529882     0.087113
   R1RR2       1.880638    3806768     0.079335
   31R32       3.714823    4561270     0.053853
   R3RR2       4.531406    4003324     0.042881
```

Writing and restoring the tethers Objects, alignments and tethers can be written to a single binary project file (see `write binary all`)
Trouble shooting `build model` may crash. A possible reason of the crash is that the pdb file is not correctly parsed due to formatting errors. Many pdb files still have formatting errors, especially those which are generated by other programs or prepared manually. In this case the `read pdb` command is trying to interpret the field shifts and, as with any guess work, frequently gets it wrong. For example, try `2ins` and you will see that the atom or residue names are shifted. To fix the problem, try to use the `exact` option of the `read pdb` command.

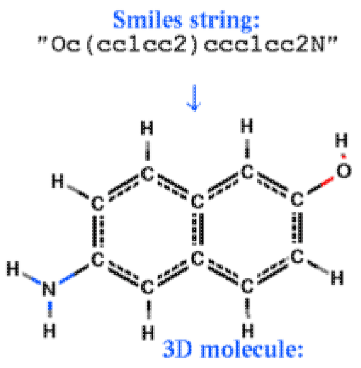
Building loop to a model by homology

`build loop` *rs_fragments*
rebuild specified loop based in a PDB-database search (see `build model`).
An example:

```
 read object s_icmhome+"crn"
 build loop a_/20:26  # rebuild this loop
```

Building object from a chemical smiles string

`build smiles` *s_smiles_string* [`name=` *s_ObjName*]



create an ICM-object from the `smiles` -string, respectively. Set `l_readMolArom` to `no` if you do not want to assign aromatic rings from a pattern of single and double bonds (and formal charge and bond symmetrization for CO2, SO2, NO2or3, PO3) upon building. To suppress the symmetrization and consequential charging of CO2, set the `l_neutralAcids` flag to `yes` .

Examples:

```
build smiles "CCO"   # ethanol

build smiles "Oc(cc1cc2)ccc1cc2N"

build smiles "Oc(cc1cc2)cc(ccc3)c1c3c2"

build smiles "C/C=C\C" # cis-2-butene

build smiles "C/C=C/C" # trans-2-butene

              # dicoronene
build smiles "c1c2ccc3ccc4c5c6c(ccc7c6c(c2c35)c2c1c1c3c5c6c"+\
             "(c1)ccc1c6c6c(cc1)ccc1ccc(c5c61)cc3c2c7)cc4"

              # NAD
build smiles "[O-]P(=O)(OCC1OC(C(O)C1O)N1C=2N=CN=C(N)C=2N=C1)"+\
             "OP(=O)([O-])OCC1OC(C(O)C1O)N=1C=CC=C(C=1)C(=O)N"

              # Hexabenzo(bc,ef,hi,kl,no,qr)coronene
build smiles "c1c2c3c4c(ccc3)c3c5c(c6c7c(ccc6)c6c8c(ccc6)c6c9"+\
             "c(ccc6)c(cc1)c2c1c9c8c7c5c41)ccc3"

              # rubrene
build smiles "c1c2c(c3ccccc3)c3c(c(c4ccccc4)c4c(cccc4)c3c3ccccc3)"+\
             "c(c2ccc1)c1ccccc1" name="rubrene"
```

Sometimes the build smiles command is not sufficient. The molecule needs to be optimized in the mmff force field and several conformations need to be sampled. A more rigorous conversion is provided by the `convert2Dto3D` macro.
See also: `Smiles` , `find molecule`.

Building object from string

`build string` *s_IcmSequence* [`name=` *s_ObjName*] [`delete`] [*i_first_amino_residue_number* (`1`)]
create an ICM-object from a *s_IcmSequence* string (see the `build` command above). To get a D-amino acid instead of L-ones simply use D as a prefix: Dala Darg. Specify N- or C-terminal modifiers directly in the file if needed. The build command will create them in some default conformation.

The `build string` command also understands short *one line* version of the full format. The short format looks like "ASD" or "ala his" and may not start from "ml " or "se ".

The possibilities are the following:

- one letter code, - it needs to be specified in upper case letters, e.g. "DD";
- full three-four letter code, e.gg. `"nter ala hise Dala cooh"`
- multiple molecules - just use a comma, a semicolon or a dot as a separator, e.g. "WWWW;AAAA;EEE" or "ala his trp; nh3+ gly coo-"
- mixed one-letter and three letter code, e.g. "AST-sep-tpo-AAA" to include phosphoserine and phosphothreonine

A list of current amino acids from the `icm.res` file is the following:

If the sequence is provided as one letter code (e.g. "ACDTCAA") or as an icm `sequence` the residue number of the first aminoacid will be set to 1 unless redefined by the optional integer *i_first_amino_residue_number*.The N-terminal residue "nh3+" will then get number 0.

Option `delete` temporarily sets the `l_confirm` flag to `no` and the old object with the same name gets overwritten.
Examples:

```
   build string "ADG-sep-HRTE" # the charged terminal groups will be added, note phosph
   build string "ADGHRTE" 2 # assign res number of 2 to 1st alanine
   build string "ADGH;RTE" # two peptides, a and b
   build string "nter ala Dhis cooh" name="pep"  # one peptide named a_pep.
   build string "ml a \nse nh3+ his coo- \nml b \nse trp" # molecules a and b
   build string IcmSequence("GHFDSFSDRT","nter","cooh") # translate and add termini
#
# Using alias  BS   build string "se $0"
   BS ala his trp
```

See also: `Sequence`, `IcmSequence`.

Building hydrogens according to topology and formal charges.

`build hydrogen` [*as_heavyAtoms*] [*i_forcedNofHydrogens*] [`cartesian`]
adds hydrogens to the specified heavy atoms according to their type and `formal charge`. All heavy atoms of the current object are used by default. If your have hydrogens already and their configuration is wrong, you can delete them with the `delete hydrogen` command. The number of hydrogens may be enforced if the optional *i_forcedNofHydrogens* argument is specified.

Option `cartesian` means that no new hydrogens are added, but, rather, the existing ones are set to new coordinates according to the heavy atoms (a better syntax for this action is `set hydrogen`).

See also the `set bond type` command, `set hydrogen` .
Examples:

```
  read mol s_icmhome+ "ex_mol"  # several small molecules
  display a_4.
  build hydrogen a_4.  # added and displayed
#
  undisplay
  display a_3.
  build hydrogen a_3.
# move one of the nydrogens
  build hydrogen a_3. cartesian # should put the hydrogen back at a correct position
```

Building molcart indices for substructure, similarity or exact search

`build molcart` {`s_tableName`|`S_tableNames`} [`sstructure`|`similarity`|`exact`]

builds (or rebuilds) various keys for molcart table.

call

`call` *s_ScriptFileName* [`only`]
invokes and executes an ICM-script file. End the script with the `quit` command, unless you want to continue to work interactively, or use it in other script.
The option `only` allows one to suppress opening the script file if the `call` command is inside a block which is not executed. By default the script file is opened and loaded into the ICM history stack anyway, but the commands from the file are not executed.
The absolute path of the script can be obtained by calling the `Path` (`last`) function.
Example:

```
  call _startup            # execute commands from _startup file
  show Path( last )
```

Example of calling scripts inside conditional expressions.

```
if Type( CONSENSUS ) != "table" then
 call _startup only # only means do not read if the table is already loaded
endif
```

center

`center` [{ *as* | `grob` }] [`only`] [`static`] [`margin=` *r_margin*]
centers and zooms the screen on selected atoms `as_` or graphics objects. Default objects: all existing atoms and graphics objects. The *r_margin* argument is given in Angstrom units and can be used to set a relative size of the selection and the frame. Normally all dimensions of the molecule/grob are taken into account, so that the molecule can be rotated without changing scale.
Options:

- `only` : **do not rescale**, translate only, i.e. move the selected atoms to the center of the graphics window
- `static` : scale only according to the visible X-Y dimensions and the margin. Do not take the Z-dimension into account in the size calculation as if you do not intend to rotate objects. That implies an assumption that the orientation of molecules/grobs/maps will not be changed.

Examples:

```
   nice "1est"
   center
   center Sphere ( a_/15:18 )
   center a_/1:2 only  # keep the scale

   read grob s_icmhome+"beethoven" # a genius
   display beethoven smooth
   center beethoven static   # 10 A margin
```

clear

`clear`

clear terminal screen
`clear selection` clear the graphical selection `as_graph`

Example:

```
nice "1crn"
as_graph = a_/1:5 # select five residues
clear selection # nothing again
```

`clear pattern` *chemarray*

clear `SMARTS` search attributes in the input `chemical array`.

Example:

```
add column t Chemical("[C;D2]")
clear pattern t.mol   # D2 attribute will be cleared
```

See also: `Exist pattern`

`clear graphic` [*os*] clears display properties , graphic representation memory and reset the graphic planes to the default.

`clear error`

clears all error and warning bits previously set by ICM. See also `Error` (i_code)

color

The color command colors different shell objects, their parts, or different graphical representations with by colors specified in various ways. The main color commands are listed below:

`color all|{wire|xstick|cpk|surface|skin|[residue|atom|variable|string] label|ribbon [base]}` *color as* `[full]`

`color` *as*|*rs*|*ms* `{molecule|object|alignment | R_values [window=`*R2_fromTo*`] } [full] [all]`

`color background|volume` *color* `# volume for the depthcuing fog color`

`color chemical` *X_chemarray* {*P_predictiveModel* | `pharmacophore` }

`color site` *ms1* `index=`*i_site*|*I_sites color_spec*

`color distance|hbond|angle|torsion` *P_distParray color*

`color` *g* `accessibility` *r_depth* `([0:1]) # occlusion coloring`

`color` *g* `[add|pseudo]` *color as* `GROB.atomColorRadius=` *r*

`color` *g* `map [`*m_valuesForColoringGrob*`]`

`color` *g*`|grob potential [ fast ] [ reverse | simple ] [`*ms_sourceAtoms*`] # electrostatic coloring`

`color` *map_Name* [*I_colorTransferFunction*] [*R2_fromTo*] [`auto`]

Options:

- `full` : allows one to set colors for atoms that are **not** displayed in addition to the displayed ones. The default only changes colors of the atoms visible in a given representation. (this option has been added in versions compiled after Sep 15, 2009). This option replaces the `set color` command for batch coloring.
- `all` : colors all graphical representations (by default it colors only the specified ones)

See also: `set color` to set atom or residue color directly and without graphics. See also: icm.clr for allowed color names and their r,g,b values; the `plot` command needs ICM colors , an `sarray` can be returned with the `Color(` *R* `)` command.

Specifying colors in ICM

There are various ways to specify a color in ICM: by name, index or `RGB representation`.

color_name | `color[`*i_index*`]` | *i_Color* | *r_Color* | `rgb=`*rgb_color*

Specifying color by name:

```
color red
```

Other color name examples: black, white, grey, blue, red, yellow, green, orange, magenta, lightblue. Color names may be observed and changed in the `icm.clr` file.

Requesting contrasting colors by index:

```
color color[4]
```

This call uses color number 4 from the list of **"named"** colors (first section of the `icm.clr` file). Colors with their numbers can be listed by the `show color` command and their total number is accessible via the `Nof( color)` function. This mode is useful if you need to color selected elements with contrasting colors rather than with a smooth spectrum.

Example:

```
  read pdb "1crn"
  display ribbon a_1crn.
  show colors
  color a_/1:5/* color[89]
  for i=1,Nof(a_/*)
    color a_/$i color[i]       # speckled coloring
  endfor
```

Specifying color by index:

```
color 3
```

Color indices are taken from the **"rainbow"** section of the `icm.clr` file. Currently there are 128 colors (i=0,127) in this section and they form a smooth transition from blue to red via white (not really a rainbow). You may change the "rainbow" colors in the icm.clr file. Number 128 becomes blue again. Using integer color indices is convenient for automatic coloring within ICM loops.

Example:

```
  display "Colors"
  for i=1,255
    color background i
    print i
  endfor
```

See also `color background example`.

Specifying colors interpolated between indexed colors:

```
color 4.5
```

The color 4.5 will be the average between the "rainbow color" 4 and "rainbow color" 5.
Specifying colors by their `RGB representation`

Color is defined as a combination of **red**, **green** and **blue** components. The triple may be specified in different formats:

`rgb` = `R_3rgb`

- as an array of 3 reals in 0..1 range

`rgb` = `I_3rgb`

- as an array of 3 integers in 0..255 range

`rgb` = `s_#rrggbb`

- as a string where each component is defined by two characters in hexadecimal form. Optionally prefixed with a hash symbol ("#").

Examples setting magenta color (mixture of red and blue):

```
color rgb={255,0,255}
color rgb={1.,0.,1.}
color rgb="#ff00ff"
color rgb="ff00ff"
```

In case the requested RGB color is not available for the graphics system, ICM finds the closest color.

Coloring molecular objects

The main color command:
`color` [*as*] *graphic_representation* [*color_spec*]

`color` [*as*] *graphic_representation* [*I_colors* | *R_colors*] [`window` = *R_2MinMax*]

graphic_representation, when specified, must be one of the following

`wire` | `hbond` | `cpk` | `ball` | `stick` | `xstick` | `surface` | `skin` | `site` | `ribbon` [`base`]

This command colors selected atoms (`as_`) or graphics object(s) according to the specified color. It is possible to either `specify a single color` *color_spec*, or provide an array (`rarray` or `iarray`) of colors to color each element of the selection according to a certain property, as electric charge or `Bfactor`.

The scale is determined by the minimal and the maximal elements of the array, independently of the array length. First the numbers in the array are scaled so that its minimum corresponds to the first color in the "rainbow" section and its maximum to the last color. Then the scaled numbers are applied sequentially to the elements of the selection. If the number of elements in the array is shorter than the number of elements in the selection, the array is applied periodically. If the color array is longer than the selection, the excessive numbers are not used for coloring but (attention!) they will be used for scaling.

The `window={` *`minValue`*`,` *`maxValue`* `}` option allows one to provide a range for color mapping. It will be used instead of the array minimum-maximum value range as the range from which the color array elements will be mapped into the rainbow colors. Moreover, values in the color array will be clamped to be in the **window** range.

In the following example the Bfactor(a_/ simple) values which may range from large negative values to large values will be clamped to the [4.,40.] range.

```
  nice "1ekg"
  color ribbon a_/ Bfactor(a_/ simple) window=4.//40.
```

Another example:

```
  read object s_icmhome+"crn"
  display a_crn.
  color a_//* Charge(a_//*) window={-1.,1.}
```

It is also possible to show a color bar in the graphics window by changing the `GRAPHICS.rainbowBarStyle` property.

Each of the command arguments has a default:

- objects *as_:* the current object (`a_`) only. **Hint:** to color all objects, use `a_*` .
- *graphic_representation:* all except ribbon. Ribbons should be colored explicitly using a `color ribbon` command.
- *color_spec.* The default coloring is by atom type, except for the **ribbon** representation which is colored by secondary structure by default.

All default values can be changed by editing the `icm.clr` file.

In DNA and RNA ribbons, bases can be colored separately (e.g. `color ribbon base a_1/* white`), the default coloring being A-red, C-cyan, G-blue, T or U-gold.

Examples of how the defaults work:

```
 nice "1crn"
 display       # also displays wire
 color         # all except ribbon colored by atom type
 color ribbon  # only ribbon of a_ by secondary structure type
 color ribbon red         # only ribbon as specified
 color a_/1:10 ribbon yellow # parts
```

More examples:

```
 build string "ASDWER"     # hexapeptide
 display
 color a_/1:4 green    # the first four residues in green
 color                 # return to default colors by atom type



 read pdb "1crn"
 display a_1crn. only
                       # color atoms according to their B-factor
 color a_1crn.//* Bfactor(a_1crn.//*)
                       # crambin's ribbon
                       # from blue N-term to red C-term gradually
 display a_/* ribbon only
 color a_/* Rarray(Count(1 Nof(a_/* ))) ribbon

                       # another crambin's ribbon
                       # from blue N-term to red C-term gradually
                       # thick worm representation
 assign sstructure a_/* "_"
 GRAPHICS.wormRadius= 0.9
 display a_/* ribbon only
 color a_/* Count(1 Nof(a_/* )) ribbon
```

Coloring 2D molecules in a chemical table

`color chemical` *X_chemarray P_model*

calculates atom contributions to the total value calculated by the *P_model* if this model is

- linear. (PLS)
- built using counted fingerprints (no external column-descriptors)

`color chemical` *X_chemarray* `pharmacophore`

color by built-in pharmacophoric definitions The list of definitions can be listed like this:

```
icm/def> show pharmacophore type
name         codesmarts                          color
--------------------------------------------------------
Negative     [Qn]C(~[O;D1])~[O;D1]               #87cefa
Negative     [Qn]P(~[O;D1])(~[O;D1])(~[O;D1])~*#87cefa
Negative     [Qn]S(~[O;D1])(~[O;D1])(~*)~*       #87cefa
Positive     [Qp][N;D3;$(N(-[*;^3])(-[*;^3])-[*;^3])]#fa8072
Positive     [Qp][N;D2;$(N(-[*;^3])-[*;^3])]     #fa8072
Positive     [Qp][N;D1;$(N-[*;^3])]              #fa8072
Positive     [Qp]C(~[N;D1])~[N;D1]               #fa8072
HBA          [Qa][O,S&v2,N&^2&X2,N&^1&X1,N&^3&X3]#98fb98
HBD          [Qd][!C;!H0]                        #ee82ee
Aromatic     [Qm]a                               #ffa500
Hydrophobic   [Qh][C&!$(C=O)&!$(C#N),S&^3,#17,#15,#35,#53]#e0ffff
```

How to color grob surface by depth

`color accessibility` *g_mesh* [*r_maxShade*]

modify the color of each surface element of a `grob` to create perception of depth. The procedure calculates for each surface element (triangle) the extent it is occluded from ambient light by other parts of the molecule, and makes the elements darker proportionally to occlusion. Thus, concave regions such as pockets become dark since the surrounding bulk of the protein blocks the light from most directions, while protrusions remain bright since they are well exposed. Repeated application of the command or using a larger *r_maxShade* (the default is 0.8) generates a more dramatic shading of the shape.

Example:

```
color accessibility g_electro 0.7
color accessibility g_electro 0.7 # do it two times for a more dramatic effect
```

To be able to come back to the initial coloring you may need to do this:

```
  clrs = Color(g_electro)
  # change grob color, e.g. with color accessibility
  color grob clrs
```

Uniquely coloring by object, molecule, residue or atom

`color` *graphic_representation* [*as_molecules*] [`object`|`molecule`|`residue`|`atom`]
a special command to color the displayed and selected molecules differently. The graphic representation field can be either empty, or one of those: `wire xstick cpk surface skin ribbon, residue label, atom label, site label, variable label` . E.g. select graphically some atoms and do this:

```
color xstick as_graph & a_*.//c* molecule
color ribbon as_graph object
color cpk as_graph molecule
color residue label as_graph residue
```

color background

`color` `background` *color_spec*

sets the background to the specified color *color_spec* in `one of the supported formats` .

Examples:

```
color background blue

color background lightyellow

color background rgb={255,255,255} # white. integers in 0..255 range

color background rgb={0.,1.,0.} # green. reals in 0.. range
```

See also: `rgb`, `color background example` .

color by alignment

`color` *as* [`wire`|`cpk`|`skin`|`ribbon`|`xstick`|`ball`|`stick`|`surface`..] `alignment`
colors specified graphics representations of the selected residues by the colors of an alignment as you see it in the alignment window of the Graphics User Interface. The color of a residue is controlled by the following factors:

- ♦ residue type
- ♦ consensus character at the residue position in the alignment
- ♦ colors as provided by the `CONSENSUSCOLOR` table.

Note that the CONSENSUSCOLOR table can be divided into sub-sections, and the active subsection can be selected from GUI.
Example:

```
  read sequence s_icmhome+"sh3"
  nice "1fyn"
  make sequence a_1  # extract 1st sequence
  group sequence sh3
  align sh3

  color a_1 ribbon alignment
  display skin white a_1 molecule
  color a_1 alignment   # colors all representations including skin
```

color grob

Color is a powerful mechanism of showing extra information on ICM `grobs` ICM grobs may have individual colors assigned to each vertex, which allows one to use grob coloring to illustrate properties of 3D surfaces.

The simplest way to set grob color is to paint it to a single color.

`color` *g_grobName color_spec*

colors the whole *g_grobName* grob to the *color_spec* color.

`color` `grob` *color_spec*

colors all grobs to *color_spec.*

Check out the `color specification` section for available *color_spec* options.

Example:

```
torus = Grob("TORUS",3.,1.)
display torus
color torus black # paint it black
color background white # this should improve the visibility
color torus rgb={127,255,212} # aquamarine, as some people call it
```

Automatic assignment of different colors to different grobs

`color grob unique`
In addition to the main `color` command which colors grobs there is a special command to automatically assign the displayed `grob`s to different colors.

See example for the `split grob` command.

Coloring grob by matrix of RGB values for each vertex.

`color` *g_grob M_rgbMatrix*
allows one to set individual colors to `grob` vertexes. Colors are specified in `RGB` format in the *M_rgbMatrix.*Each row of the matrix is an RGB triple. This type of matrix may be obtained by the `Color(` *g_grob* `)` function.

Examples:

```
torus = Grob("TORUS",3.5,0.5)
display torus smooth
n = Nof(torus)
R_rgb = Count(1 n/2)/Real(n/2) // Count(n-n/2 1)/Real(n-n/2)
add matrix M_rgb R_rgb
add matrix M_rgb Rarray(n,0.3)
add matrix M_rgb Rarray(n,0.7)
color torus Transpose(M_rgb)
```

This command allows one to create special effects, like gradual disappearance of a grob into background:

```
# set the scene
color background black

# uncomment these lines to get a more sophisticated example
# torus = Grob("TORUS",3.5,0.5)
# display torus smooth
# color torus blue

# the active grob
g = Grob("SPHERE",3.,5)  # a wire sphere
display g smooth
color g Random(Nof(g),3, 0., 1.) # color randomly
M_colors = Color(g)               # extract current colors
# make the sphere disappear (modern poetry)
for i=1,20    # shineStyle = "color" makes it disappear completely
 color g (1.-i/20.)*M_colors
endfor
for i=20,1,-1 # bring the sphere back
 color g (1.-i/20.)*M_colors
endfor
```

Coloring grob by proximity to atoms

`color` *g_grobName as_closeAtoms color_spec* [`add`|`pseudo`]

colors vertices of the *grob* which are less than `GROB.atomSphereRadius` to any of the selected atoms. The default value for the radius is 4Å.

Options:

- ♦ `add` : adds van der Waals radius for each atom to the `GROB.atomSphereRadius` parameter
- ♦ `pseudo` : for hydrogen bonding acceptors considers distances from LONE-PAIR centers at 1.7A distance from the acceptor atoms. If an atom is not an acceptor, the atom itself is

considered. Note that a_//HA is a selection for hydrogen bonding acceptors and a_//HD is the donor selection.

Example in which we color 1.3 radius sphere around the lone pairs of hydrogen bonding acceptors:

```
color a_REC.//HA g_pocket magenta pseudo  GROB.atomSphereRadius=1.3
```

See `color specification` for the definition of *color_spec*.
See also: `Grob( g R_6)` function to return a patch of certain color.

Example:

```
nice "1crn"
make grob skin a_1crn. name="g_1crn"
display g_1crn
color g_1crn green
color g_1crn a_1crn.//1:60 red # color a patch by atom proximity
```

See also: `make grob skin`, `make grob potential` .

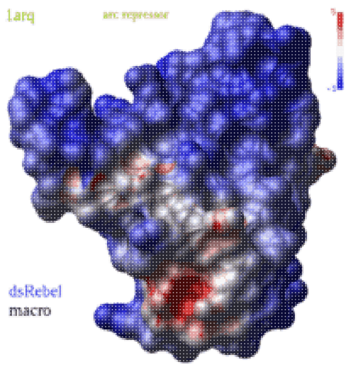
Coloring surfaces by 3D scalar field

`color` *g_grob* `map` *map_Name I_transferFunction R_2mapValueBounds* [*color_spec*]
colors vertices of the *g_grob* by the values of the *map_Name* . The map values at each grid point are first clamped into the *R_2mapValueBounds* range, then this range is divided according to the number of elements in the transfer function and each point is colored according to the value of the transfer function. The optional *color_spec* parameter is explained in the `color specification` section.
The new color will be *mixed* with the current color of grob points. Therefore if you want to color each of 3 RGB channels with a different normalized property value, first color the grob black, and then color with the `red` , `green` , or `blue` color depending on which channel you intend to use. Note that zero in the transfer function correspond to *no color* . Corresponding grob nodes will not be colored.
Transfer function is the same to the one in `color map` but has certain differences. This function (e.g. `{0 0 0 1 2 3}`) contains any number of positive integers. 0 means "do not color", and each positive value is a scaling factor for the *color* provided as an argument, or a parameter to select a color from a predefined rainbow. In the above example, the *R_2mapValueBounds* range will be divided into 6 ranges and each value range will be colored accordingly.
Example in which we color the vertices of a grob by inverted values of truncated hydrophobic potential:

```
  read obj s_icmhome+"data/xpdb/1sre.ob"
  display a_
  make grob skin a_2 a_2 name="g_pocket"        # create g_pocket
  make map potential "gs" Sphere( g_pocket a_1)
  compress g_pocket 1.
  color g_pocket black
  color g_pocket map -m_gs { 0,0,0,3,4,5 } { 0. 0.5  } green
  display g_pocket
  h = Transpose( Color( g_pocket ) )[2]  # extract hydrophobicity
```

Coloring grob by electrostatic potential

`color` *g*|`grob` `potential` [`fast`] [`reverse` | `simple`] [*ms_sourceAtoms*]



(REBEL feature) calculates electrostatic potential `waterRadius` away from the surface of the *g_skin* graphics object and color surface elements according to this potential from red to blue. **Important** the location of the center of the water probe is determined by the grob normal (you can change it with the `set g_ reverse` command). If you compute the potential at a blob outside the molecule but with the normals point outwards, use the `reverse` option. To compute potential without any positional correction including normals use the `simple` option.
The potential is calculated either by the `REBEL` boundary element solution of the Poisson equation, or, if option `fast` is specified, by a simple Coulomb formula with the `dielConstExtern` dielectric constant (78.5 by default).

The local value of potential is clamped to the range [`-maxColorPotential, +maxColorPotential`]. It means that a potential larger than `maxColorPotential` is represented by the same blue color, while values smaller than `maxColorPotential` are represented by the same red color. The real range is reported by the command and you can adjust `maxColorPotential` to cover the whole range. To suppress the absolute maxColorPotential threshold and use auto-scaling instead set `maxColorPotential` to 0. The color bar with values will appear according to the `GRAPHICS.rainbowBarStyle` preference. There are two macros to generate potential-colored skins: `rebel` and `rebelAllAtom`
The second one (given below) considers all the atoms (including hydrogens) with their charges. The mean value of the potential at the surface is returned in `r_out` , and the root mean square deviation of the potential is return in `r_2out` shell variables, respectively. The averaging is free from bias due to uneven density of grob points. It uses equal size cubes distributed evenly over the surface. The number of representative cubes used for the calculation is return in `i_out` .

Examples:

```
read object s_icmhome + "crn"
display a_1
make grob skin a_1 name="g_crn"
make boundary a_1
display g_crn
color g_crn potential
```

See also: `electroMethod`, `make boundary`, `delete boundary`, `show energy "el"`, `Potential()`.

color label

`color label` [*as*] *color_spec*

`color label` *as* [*I_colors* | *R_colors*]
Colors labels associated with the selected residues or atoms. A simple option is to specify a single color using `color specification` formats. It is also possible to provide colors for each atom using an `iarray` *I_colors* or `rarray` *R_colors*If no atom `selection` is specified, all labels are colored.
Examples:

```
 read object s_icmhome + "crn"
 display a_//n,ca,c white
 display label residue
 color label a_/* Count(1 Nof(a_/*))
 #
 color label a_/5:10 magenta

 read object s_icmhome + "crn"
 display a_//n,ca,c white
 display label residue
 color label lightyellow
```

See also: `display label`, `color object`, `resLabelStyle` .

color map

`color` *map_Name* [*I_colorTransferFunction*] [*R2_fromTo*] [`auto`]
color the current or the specified map according to the color transfer function supplied as *I_colorTransferFunction.*
The default: By default the maps are colored in such a way that points with zero map values become transparent while values above and below zero are colored by shades of blue or red, respectively.
The *R2_fromTo* array of two elements allows one to set the lower and the upper boundaries for the red and blue colors, respectively. All values above and below will be trimmed to the range. For electrostatic maps the array is set to `-5.,5.` by default.
In the `auto` mode all grid points are divided to Nof(*I_colorTransferFunction*) color classes according to the normalized function value (sigma units around the mean value) and each class is colored as specified in the *I_colorTransferFunction* (0 means transparent).
If the number of *I_colorTransferFunction* elements is odd (2* *n+1*) the class boundaries are the

following:

- ♦ -infinity
- ♦ Mean- *n* *sigma,
- ♦ Mean-(*n* -1)*sigma,
- ♦ Mean-(*n* -2)*sigma,
- ♦ ...
- ♦ Mean- *1*sigma*,
- ♦ Mean
- ♦ Mean+ *1*sigma*,
- ♦ ...
- ♦ Mean+(*n* -1)*sigma,
- ♦ Mean+(*n*)*sigma.
- ♦ +infinity

For even number of elements (2* *n*), boundaries are shifted by half a sigma, so that the middle class is between Mean-0.5*sigma and Mean+0.5*sigma. Color codes are in arbitrary units since the array is normalized so that the highest value corresponds to the red color. Deep blue is 1. Zero is always the transparent color (no coloring). The spectrum is defined in the `icm.clr` file.
Examples of coloring:

- ♦ `{0 0 0 0 0,0 0 0 3 10}` default map coloring, color only high densities (blue from 3 to 4 Sigma, red >4 Sigma). Comma only shows you where the mean is.
- ♦ `{0 1 0}` color only Mean+- 0.5*sigma nodes, ignore high and low densities.
- ♦ `{1 0 2}` color low and high densities by different colors, ignore densities around the mean.
- ♦ `{1 2 3 0 5 6 7}` similar the previous one, but with more grades

Examples:

```
read pdb "1crn"
make map potential name="mpot"
color mpot {1 2 0 4 5}
# OR
color mpot
```

color volume

`color volume` *color_spec*

determines the color of the `fog` in the depth-cueing mode (activated with `Ctrl-D`). Format of *color_spec* is explained `here`.

For example, if you want that distant parts of you structure are darker (black fog), but the background is sky-blue, you will do the following:

```
color background lightblue
color volume black
```

compare: setting conformation comparison parameters for the montecarlo command

`compare` *vs* | *as* [`static` | `chemical` | `surface`] | [`compareMethod=..`]

sets a metric for calculating a distance between different `conformations` in a `stack` .
The goal of the two following `compare` commands is to provide a desired **setting before the `montecarlo` command** and stack operations. This command defines a filter which is used to decide how many and what conformations from the stochastic optimization trajectory are kept as low energy representatives of a certain area in conformational space. This metric is also used for the subsequent **stack** manipulations, e.g. `compress stack`.
The `compare` command defines the distance measure between molecular conformations which is used to form a set of different low energy conformers in the course of the stochastic global optimization procedure. The defined distance is compared with the `vicinity` parameter and determines whether two conformations should be considered different or similar (i.e. belonging to the same slot in the `conformational stack`). The compare command determines the spectrum of conformations that will be retained in the stack, accumulated during a `montecarlo` procedure. The default comparison set is a set of all free torsion variables (see `compare vs_`). `Other methods` compare atom RMSD with and without superposition, using chemical

superposition, and compare only the atoms in the interface with a molecule (`compare surface`).

Please note that the compare command can change the `compareMethod` preference. Example:

```
  montecarlo v_//2 compareMethod ="chemical static"  # suitable for docking
```

See also `montecarlo`, `compareMethod`.

Compare by deviations of cartesian coordinates with or without superposition

`compare` [`static`] *as*
The command needs to be run when Cartesian root-mean-square deviation for positions of selected atoms (*as_*) as a distance measure between stack conformations. Set the `vicinity` parameter to about 2.0 Angstrom if you want to consider conformations deviating by more than 2 A as different conformational families.
By default the selected atoms in different conformations will be optimally *superimposed before* the coordinate RMSD is calculated. The `static` option suppresses superposition and measures absolute deviation of the coordinates between conformations. The `static` option is relevant for ligand atoms in docking simulations to a static receptor.
The result of this procedure is that an internal flag is set to perform cartesian RMSD calculations during `montecarlo` run, and a set of selected atoms is marked for comparison.

Compare by deviations of internal coordinates/torsions.

`compare` *vs*
use angular root-mean-square deviation for selected internal variables (usually torsion angles) as distance (set `vicinity` to at least 30.0 degrees accordingly)
Examples:

```
   compare v_//phi,psi           # compare ONLY the backbone angles
   vicinity=30.0                 # consider two conformations
                                 # with phi-psi RMSD < 30. as similar

   compare a_2//ca static        # compare Cartesian deviations
                                 # of the second molecule's alpha-carbon atoms
                                 # without prior optimal superposition
   vicinity=3.0                  # consider two conformations with second
                                 # molecule deviation < 3 A as similar
```

Compare by coordinate deviations of the surface patches only

Compare by surface patch rmsd: dynamically selecting comparison atoms

`compare surface` *as_currentObjSelection* | *as_staticReferenceObject*.
Similarly to `compare static` *as_* it will look at absolute deviations of coordinates, but the comparison will be applied dynamically only to a **patch sub-selection** of the atoms in the current object in the `selectSphereRadius` (default 5. A) proximity to the non-current-object atoms of the *as_* selection. The selection typically would look like this: a*_activeIcmObject.*//ca | a*_staticPdbReceptorObject.*//ca
Example:

```
 compare a_runObj.//ca | a_recName.//ca surface
```

Note that this command **dynamically** calculates a subset of *as_currentObjSelection* **near** *as_staticReferenceObject* . This distance (static RMSD) is used inside `montecarlo` command or in `compress stack` .
The surface mode is useful for protein-protein docking simulations when you want to measure the `sRmsd` distance between the current conformation and the stack conformations ONLY for the interface residues of the moving molecule. The interface residues are dynamically determined as those which are close to the static receptor specified in the second part of the selection. This static receptor should reside in a separate object.
The *vicinity* size is determined by the `selectSphereRadius` parameter
An example in which we sRmsd-compare only those carbons of barstar which are next to the barnase surface.

```
 read pdb "1bgs"     # a complex
 read pdb "1a19.a/"  # the protein ligand only
 convert
...     # make maps and other actions to prepare protein-protein docking
 compare a_//c* | a_1.1 surface  # will use only
 selectSphereRadius = 7.
...
 montecarlo
```

compress

compress grob vertices, shell objects, or stack conformations

compress graphical objects

`compress` *g_grobName1 g_grobName2* .. [*r_minimalEdgeLength*=`.5`]

`compress` `grob` [`selection`] [*r_minimalEdgeLength*=`.5`]
simplify a grob (graphical object) by eliminating/merging small triangles into bigger ones. This procedure allows one to generate very "low-resolution" molecular surfaces. The default value of the *r_minimalEdgeLength* is 0.5 Angstroms. Typically compression with the 1. A minimal edge parameter reduces the number of triangles by an order of magnitude. The compression algorithm does not change the connectivity of the surface. Therefore you can still `split` the compressed grob and find the fully enclosed cavities.
The compress command returns the new number of verteces in `i_out` and the new number of triangles in `r_out` variables, respectively (for the last compressed grob only).
Example:

```
read pdb "1crn"
make grob skin smooth name="g_1crn" # creates a grob with many triangles
display g_1crn
compress g_1crn 1. # significantly reduces the number of triangles in the grob
display g_1crn
compress g_1crn 4. # further simplification of the grob
display g_1crn
```

It is important in this example to use the `make grob skin` command with the `smooth` option, since it closes the cusps.

See also:

- `delete all compress` # to delete all objects/grobs/maps not used in slides
- `compress binary` *file.icb* # compresses .icb file files

compress stack of molecular conformations

`compress` `stack` [`fast`] [*i_fromConfNumber i_toConfNumber*] [*r_enerDiff*]

Remove similar and/or high energy conformations from the `conformational stack`.
During a `montecarlo` run, some conformations of the generated `conformational stack` may be substituted by newly calculated ones with lower energies. New conformations may violate the initially correct distribution of the conformations in the slots of the stack as defined by the `vicinity` parameter and by comparison mode specified by the `compare` command. The **compress** command compares all the pairs of the stack conformations, identifies pairs of conformations in which two conformations are separated by a distance less than the `vicinity` threshold, and removes the higher energy stack conformation from each close pair. Optional arguments *i_fromConfNumber* and *i_toConfNumber* define a subset of the conformations in the stack which are to be analyzed and compressed (if any). The whole stack (from the first to the last conformations) is processed by default.
Note that if two close conformations are compressed into the better energy one, the number of visits of the resulting conformation will be a **sum** of the two numbers of visits.
The `fast` option applies an iterative compression algorithm which can be several orders of magnitude faster but the result may slightly differ form the default compress. The fast algorithm algorithm performs the following steps:

1. sort conformations by energy
2. start from the lowest energy conformation

3. find all conformations with higher energy than the current conformation within `vicinity`.
4. delete similar conformations with higher energies and compress stack
5. move to the next conformation in the new sorted stack, make it current and go back to step 3

See also `How to merge and compress several conformational stacks`
Example (define a distance and compress) we generate two stacks, merge them and re-compress two sets with a different comparison criterion:

```
build string "VTLFVALY"
mncallsMC = 5000
montecarlo   # generates stack
write stack "f1"
delete stack # clean up and
montecarlo   # generates another stack
read stack append "f1"  #
compare v_/2:5/phi,psi  # compare settings are different
vicinity = 40.          #
compress stack fast
vicinity = 20.          # new vicinity
compress stack
compress stack  2.0  # remove confs > 2 kcal/mole higher than the lowest one
```

See also: `compress binary`

---

compress files from ICM

`compress binary` *s_inputfile* [`filename=`*s_gzipfile* | `delete`]

Compresses the *s_inputfile* file using GZIP algorithm. If the `filename` is specified, the compressed file will be saved as *s_gzipfile.*If the `delete` option is specifeid, the compressed file replaces the input file (*in place compression*). Otherwise (by default) .gz extension is added to produce the compressed file name.

Example:

```
read pdb "1crn"; make map potential name="x"; write map x # create x.map file
```

```
compress binary "x.map" delete # compress in place
```

See also:

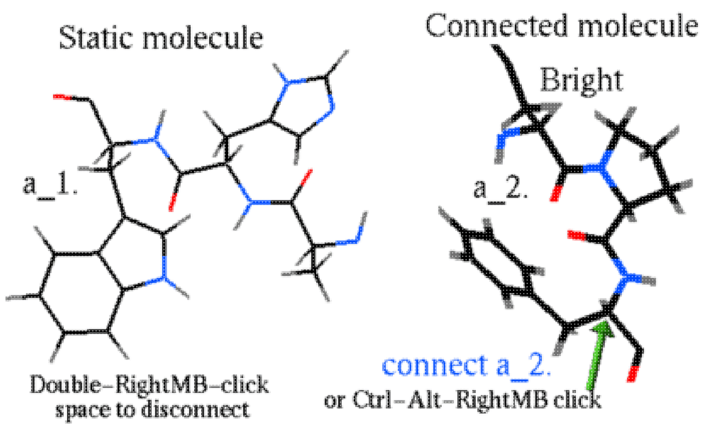
- `delete sequence` `compress`
- `delete all compress` # leave only objects in slides
- `compress grob`

connect

`connect` [`append`] [`none`] [*ms_molecule* | *g_grob*]

`connect` `none`

connects selected molecules to the mouse for independent rotation (by the LeftMouseButton) and translation (MiddleMouseButton) with respect to the original coordinate frame.



Option `append` will add selected molecule to the previously connected molecules
Note, that rotations/translations in the connect mode actually **change the atomic coordinates** of the selected molecules and keep the coordinate system unchanged in your graphics window.
To restore the usual global mode (i.e. all objects/molecules are disconnected and the mouse does not change their absolute positions, but rather the point of view), hit the Esc key when the cursor is in the graphics window. To restore the global mode **temporarily** press the `Shift` button.
Use: `connect none` to switch back to the global connection Examples:

```
read pdb "1eff"
copy a_1eff. # create something else in the scene
display ribbon a_*.
connect a_1eff.
# move it around now
connect none # disconnect
```

Connect to a Mysql database or database file

`connect molcart` {*S_host_user_pass_db*|*s_host s_user s_pass s_db*} [`name=`*s_connectionID*]

Connects to the database server specified by the command parameters. It is possible to also specify the *s_connectionID* which will be assigned to the connection. Parameters returned by the `Name(sql connect)` may be used in this command.

`connect molcart on`

Reconnects to the current Molcart.

`connect molcart refresh`

Reconnects to Molcart using settings stored in user's preferences.

`connect molcart filename=`*s_file* [*s_db*] [`name=`*s_connectionID*]

Opens a `Molsoft database file`. Database name *s_db* and the *s_connectionID* may be specified.

`connect molcart` *s_connectionID* `off`

Disconnects specified Molcart connection. See `molcart connection options` for explanation

`connect molcart local off`

Closes all open `database files`.

See also: `molcart`, `molcart connection options`, `list molcart`, `set molcart`, `Name molcart`.

continue

`continue`

skip commands until the nearest `endfor` or `endwhile` .
Example:

```
for i=1,5
  if i==3 continue  # do not print 3
  print i
endfor
```

See also: `flow control statements`.

convert

`convert` [*os_nonICM*]
[`auto`|`charge`|`exact`|`heavy`|`graphic`|`selection`|`simple`|`tether`|`selftether`|`tree=`*s_smiles*
[*s_objName*]

`convert` *as_icmRootAtom* [`sstructure=`*as_scaffold*] [`auto`] # `root the icm-tree at` *as*

`convert` *rs_patches* ..
the first convert command converts an incomplete non-ICM-object (e.g. object of type 'X-Ray' resulting from the `read pdb` command) into a true ICM-object for which you may calculate energy, build a molecular surface and perform all operations.

Options:

- `auto` - convert in place, preserve `graphics` and `selections`, e.g. convert auto selftether
- `charge` - transfer charge from the original
- `exact` - do not use the icm.res library by res name, convert as is.
- `heavy` - regularization (obsolete)
- `graphic` - transfer graphical attributes
- `selection` - transfer selection (as_graph)
- `simple` - special mode for disjoint chemicals
- `tether` - impose tethers to the original (use `selftether` for in-place or auto mode)
- `tree=` *s_smiles* - build the tree according to the smiles topology (small mol. convertion)
- `selftether` - imposes selftethers to the original coordinates, `set field` for the added heavy atoms ("_ADDED") and shifted upon conversion atoms ("_SHIFTED"), e.g. `display cpk Select( a_// "_ADDED")`

Description
There are two principally different modes of conversion. In the default mode the program looks at the **residue name** and tries to find a full-atom description of this residue in the `icm.res` file. This search is suppressed with the `exact` option.
Hydrogen atoms will be added if the converted residues are known to the program and described in the `icm.res` library. If the object selection is omitted, the `current object` will be converted. If default *s_newObjectName* is generated by adding number `"1"` to the source object name. If *s_newObjectName* is the same as a name of the input object, the input object will be overwritten. (in-place conversion)

The default `convert` command is best used to convert `PDB` entries which have explicit residue descriptions and usually do not have hydrogen coordinates. In this mode each residue name is searched in the icm.res file and the coordinates of the present heavy atoms are used to calculate the internal geometrical variables (bond lengths, bond angles, phase and torsion angles) for the full atom model.

Every ICM atom will store the original coordinates as `selftether` (try show a_// and watch for the `ts=` *x* , *y* , *z* record. Later these selftethers can be used with the "ts" term.
The `exact` option: converting protein with unusual amino-acids
Some pdb-entries may contain non-amino acid residues, or *modified* amino-acid residues which do not need to be replaced by standard full atom library entries with *the same name* . In this case use the `exact` option. This option suppresses interpretation by short residue name and converts the **existing** atoms and bonds in single-residue molecules (amino acids in peptides and proteins will

still be extended by hydrogens upon conversion, to suppress that conversion write the molecule as mol and read it back, then `convert exact`). Option exact may be necessary because chemical compounds with a four-letter short name identical to one of the amino-acid residues, could be mistakenly converted into an amino-acid with a corresponding name.
The `charge` option
Normally, upon conversion, the atomic charges are taken from the `icm.res` library entries. Option `charge` tells the program to inherit atomic charges from the *os_non-ICM-object.* For small molecules, use `set charge`, `set bond type` and, possibly, `build hydrogens` before conversion of a new compound. `i_out` will contain the number of heavy atoms missing from the pdb-template.

The `graphic` option preserves the graphical representations and colors as is.

The `selection` option preserves the atom selection bit during the conversion. Useful for in-place convert.

The `sstructure=` *tree_substructure* option makes sure that the tree is drawn through the substructure. It also needs a consistent entry atom provided as *as_newRoot* argument.

The `auto` option converts in-place preserving graphics and selection information. This is a convenient shortcut for the following combination:

- `graphic`
- `selection`
- *s_newObjectName* is set to the input object name

The `smiles` option (an addition to `auto` option) allows one to explicitly derive a tree structure from the `smiles` string. If the smiles string matches only part of the molecule then the rest of the tree will be built according to the default rules.

Additional cleanup
Actually more procedures need to be performed to prepare a functional object from crystallographic coordinates, e.g. identifying optimal positions of added polar hydrogens, assigning the most isomeric form of `histidine` , and finding a correct orientation of side-chain groups for glutamine and asparagine.
We recommend the `convertObject` macro instead of the plain `convert` command to achieve those goals.
Refining the model
To refine a model use the `refineModel` macro.
`convertObject` macro
The `convertObject` macro is a convenient next layer on the convert command. The macro may convert only a few molecules out of your pdb file, optimize hydrogens and do some other useful improvements of the model.

Output.

- the converted object
- `i_out` : the number of heavy atoms missing from the pdb-template,
- `r_out` : rmsd from the pdb-template atoms (non zero for residues with bad coordinates),
- `i_2out` : the number of deviating by more than 0.2A atoms heavy atoms,
- `r_2out` : the maximal deviation

Selection tags (`Select` (*as tag*) returns the selection) :

- "built" -heavy atoms that were missing in a pdb (e.g. some lys and arg in 1qz5)
- "shifted" -atoms that shift after conversion (e.g. silly lysines in 1qz5)

Example:

```
read pdb "1crn"
display
as_graph = a_//c*
convert auto    # converts in-place preservinf slection and graphics
strip virtual
convert         # creates new a_1crn_1. object
```

If single atom is provided as an input selection it will be taken as a new ICM tree root. See `convert and reroot` for details.

See also:

- `strip`
- `convertObject`
- `convert2Dto3D`
- `set cartesian`
- `selftether`
- `Select` (*as s_tag*)

Comparing convert, minimize tether and regularization.

It is important to understand the difference between the **convert** command, the `minimize tether` command and the `regularization procedure` implemented in the macro `regul`
.
All three create ICM-objects from PDB coordinates, but details of generated conformations and the amount of energy strain will differ.
We recommend to use `convertObject` macro for most serious applications involving energy optimization.
convert

- uses all-atom residue templates (including hydrogens) from the icm.res library
- creates temporary ICM-library descriptions for unknown residues
- makes geometry **identical** to the PDB coordinates: bond length and bond angles may be distorted.
- the converted structure will be energy strained because of common imperfections of the PDB entries and the hydrogen atoms added by the procedure
- C-alpha-only structures will not be properly converted because a special prediction algorithm is required to extrapolate the coordinates of all atoms from `C-alpha` atom positions.
- these objects are good enough for graphics, skin, secondary structure assignment, rigid body docking. They are not good for loop modeling and side-chain modeling.
- needs to be followed by polar hydrogen placement and histidine state prediction (implemented in the `convertObject` macro)

`minimize tether` threading a regular polypeptide through an incomplete/gapped set of coordinates.

- you need to create a `sequence file` first and use the `build` command;
- you will need to create the missing residues manually, say, with the `write library` command;
- build will use all-atom residue templates including hydrogens, and will preserves the fixation;
- the linear chain with *fixed idealized covalent geometry* or, actually, any fixation you define, will be threaded onto the PDB coordinates in the best possible way;
- Ca-atom PDB structures will be handled properly if all backbone torsion angles are unfixed;
- the resulting ICM-object will be strained and will need further relaxation.

full `regularization` and refinement

- uses `minimize tether` to create the starting conformation;
- employs a multi-step energy minimization (annealing) of the structure to relief energy strain;
- these are the best objects that can create in ICM for further simulations.

(see macro `regul` for details).
Examples:

```
read pdb "1a28.a/"           # reading just the first molecule
convertObject yes yes no no  # the best way to prepare for docking
                             # convert + optimizes polar H, His and Pro

read pdb "1crn"         # X-ray object, no hydrogens, no energy parameters
convert                 # a_1crn_icm ICM-object will be created
convert a_1. "new"      # a_new. ICM-object will be created
convert a_1. exact      # keep modified residues as is

read mol2 s_icmhome+"ex_mol2"
set object a_catjuc.
build hydrogen
set type mmff
set charge mmff
convert
```

Creating a multi-part molecule in which parts are separately controlled.

If you want to create a local "epitope" of a protein with chain fragments around a particular area, it can be done with the

`convert` *rs_fragments*

command. This command will create a molecule divided into fragments and each fragment will start from virtual atoms `vt1` and `vt2` and will be controlled with 6 virtual variables. The first vt1 of the second, third etc. fragments will be connected to the first real atom of the first fragment. Example:

```
read pdb "1crn"
convert a_/4:10,12,27:33,41:45
Nof( a_m//vt1 ) # the number of pieces
show v_/P1/V  # the pos. variables of the 1st part
show v_/P2/V  # the pos. variables of the 2st part
display ribbon
color ribbon a_/P3  # showing the 3rd part
```

This operation is useful to create a local patch object for docking of global optimization.

Converting a chemical compound from a mol/sdf or mol2 files.

To convert a chemical from GUI menus, follow these steps:

- make sure that bond types and formal charges are correct
- select the `MolMechanics.ICM-Convert.Chemical` menu item, check the parameters and press OK. Normally to convert from 2D to 3D you need to optimize the ligand. ICM will perform a multiple start global optimization using the MMFF94 force field (internally it runs the `convert2Dto3D` macro). If you want to preserve the geometry, select the `keepGeometry` option.

Command line conversion To perform the same conversion in a *batch* run the `convert2Dto3D` macro, or, to make a conversion without full optimization from a command line or script, issue the following commands:

```
# assuming that bond types and formal charges are correct
build hydrogen
set type mmff
set charge mmff
randomize a_//!vt* 0.01 # sometimes it helps to avoid singularities
convert
set v_//T3 180.  # making flat peptide bonds
fix v_//T3       # optional
```

Example of geometry optimization:

```
read mol input = String( Chemical("C(C(O)=O)N1C(C(=Cc2ccc(c3ccccc3[Cl])o2)SC1=S)=O"))
convert2Dto3D a_ yes yes yes yes
list convert2Dto3D
```

Converting a chemical compound and rerooting the tree at the same time

`convert` *as_rootAtom* [`auto`]
if an atom selection is provided instead of the object selection, the tree will be rerooted to the selected atom. The converted molecule will have the *as_rootAtom* located at the root of molecular tree so that it is convenient to `modify` another molecule with the converted molecule.

`auto` option behaves as in normal `convert` command. It preserves selection and graphics and preforms in-place conversion.

If you need to reroot an ICM object, do the following:

- strip it to a non-ICM object: e.g. `strip virtual`
- re-root and convert, e.g. convert a_//hb1 .

Example:

```
build smiles "C(=CC=C(C1)C(=O)O)C=1"
display wire
wireStyle = "tree"
strip a_ virtual
convert a_//h31 auto   # converts from a new root
```

copy

copies stuff which CANNOT be copied by direct assignment such as: `a=b`

`copy` *os* [*s_newObjectName*]
[`delete`|`display`|`graphic`|`selection`|`stack`|`strip`|`tether`]

creates a copy of *os_* with the specified name. Default source object is the current object. The default name is `"copy"` (object `a_copy.`)
Options:

- `delete` forces the command to overwrite the object with the same name if there is a name conflict.
- `display` or `graphic` copy the display attributes of the parent
- `selection` copies named *selections* defined on the parent object into the copy
- `stack` copies internal stack of the object (see store-object-stack) (the stored stack is not copied by default)
- `strip` applies the `strip` operation to the copied object. The stripped object has a PDB type and is much smaller in memory.
- `tether` applies tethers from the source object to the atoms of the copy-object. For further refinement see the `refineModel` macro.

Examples:

```
 read pdb "1crn"
 copy a_          # creates a_copy.
 copy a_1. "aaa"  # creates a_aaa.

 read object s_icmhome+"crn" # read ICM object
 copy a_ strip delete tether # create a_copy. and tether to it
```

crypt

`crypt key=` *s_password* { *s_fileName* | `string=` *s_string* }
encrypts the file *s_fileName* or string *s_string* in place (the size of the encrypted file/string is exactly the same), adds extension .e to the file name. If string is encrypted, its name is not changed. Apply the operation again to restore the file or string. You may encrypt both text and binary files. Note that this command has nothing to do with the unix crypt utility. ICM uses different algorithm.
Examples:

```
crypt key="HeyMan" "_secretScript"    # encrypt and create *.e file
crypt key="HeyMan" "_secretScript.e"  # decrypt it



ss="Secret rumour: Div(Rot(F))=0 !"
crypt key="fomka" string = ss  # encrypt
show ss
crypt key="fomka" string = ss  # decrypt
show ss
```

Date data-type

A basic ICM class for arrays of date objects

See also: `Date`.

delete ICM shell objects

delete shell objects or their parts.
delete ICM-shell object

`delete` [`alias`] [`alignment`] [`factor`] [`grob`] [`iarray`] [`integer`] [`logical`] [`macro`] [`map`] [`matrix`] [`profile`] [`rarray`] [`sarray`] [`sequence`] [`string`] { `name1` | *s_namePattern1* } `name2` ...

`delete all` # `to delete all shell objects not marked with a no delete flag`

ICM-shell objects have unique names; to delete some of them just type
`delete` [`mute`] { *icm-shell*-`objectName1` | *s_namePattern1* } *icm-shell*-`objectName2` ...
You may use name patterns with **wildcards** (see `pattern matching`) and add explicit specification of the ICM-shell object type, if you want the search to match only the objects of particular type. If the ICM-shell object type is not specified, all the shell-variables will be considered.
Option `mute` will temporarily switch off the `l_confirm` flag.
delete class

`delete string` *className*

`delete string command` | `html`

to delete icm-command files or html-documents loaded into ICM

`delete rarray view`

to delete all the views (returned by the `View`() function)

Examples:

```
   delete aaa           # delete ICM-shell object aaa
   delete a b c         # delete ICM-shell objects a, b and c
   delete "*"           # delete ALL ICM-shell objects added by user
   delete "mc?a*" mute  # delete ICM-shell objects matching the pattern
   delete rarrays       # delete ALL real array
   delete objects       # delete ALL molecular objects, same as delete a_*.
   delete rarray "a*"   # delete real arrays starting with 'a'
```

Deleting array elements To delete a selection of array elements specified as an index expression or an integer array of indexes, use the expression from the following example:

```
  a={1 2 3 4 5 6}   # we want to delete elements from it
  a=a[2:4]          # retain only elements 2:4
  a={1 2 3 4 5 6}
  a=a[{2,3}//{5,6}] # retain only 4 elements elements
  a=Count(100)
  a=a[Count(1,10)//Count(21,30)]  # retain ranges 1:10 and 21 to 30
```

Deleting table elements table rows can be deleted directly with the delete command, e.g.

```
 delete t.A>1
 delete t[{1 3 5}]
```

---

delete alias

`delete alias`
see `alias delete` *`alias_name`* . Example:

```
alias ls list
alias delete ls
```

Delete from database

`delete molcart table` *s_dbtable* [*connection_options*]

Deletes table from `Molcart` database with all index tables, related indexes and metadata. Database connection may be specified by *`connection_options`*

Delete plots from the table

`delete plot` *table* [`name=`*s_handle*]

This command deletes from the table all plots or only the plots with the specified name (see `make plot`).

ICM table plots are stored in the table header as an `sarray` *T.plot*, so 'delete plot T' is identical to 'delete T.plot'

delete selection variable

`delete as_selectionName`
or
`delete` *vs_selectionName*
delete named variable with atom or v_ selections. The number of named selections is limited to about 10 in each category, therefore you may need to delete them from time to time.
Important: keep in mind that deleting the named selection is not the same as deleting actual objects, molecules or atoms selected by them. To delete atoms selected by a named variable in an non-ICM object, add keyword `atom` (see `delete atom nameSelection`)
Examples:

```
 build string "ASFGD"      # build a molecule
 vsel = v_//phi,psi    # this is a vselection
 delete vsel

 asel = a_//c*,n*      # this an aselection (atom selection)
 delete asel           # delete variable asel, do not touch the atoms
 delete atom asel      # delete atoms in a non-ICM object
```

Delete array elements

`delete variable` *array* {*i_elementNumber*|*I_elementNumbers*}

delete one or more elements from any array. If the array is a column in a table *T*, use the `delete T[i]` command which can delete both a single row, e.g. `delete t[2]`, or a row selection.

Examples:

```
a={1 2 3}
b={1. 2. 3.}
c={"a" "b" "c"}
delete variable a 2  # deletes the 2nd element of the array
show a
 {1 3}
delete variable b 2
delete variable c 2
#
a = Count(100)
delete variable a Count(50)*2 # deletes even numbers
```

`delete variable` *pairdistArray I_pos*

Removes elements at positions *I_pos* from the array

delete atom

`delete` *as_atoms*

`delete atoms` *as_namedSelection*
delete selected atoms in a non-ICM object. The selection here must be a constant atom selection, rather than a named selection (e.g. you can say `delete a_/1:10/*` but NOT `aaa = a_/1:10/*` , `delete aaa`).
To delete a named variable, use `delete atom` *name* Example:

```
  read pdb "1crn"
  delete a_/1:10/*

  aaa = a_/18:20
  delete atom aaa
```

See also: `build atom` , `delete hydrogen`

---

delete directory

`delete directory` *s_Directory*
delete directory. Example:

```
delete directory "/home/doe/temp/"
```

See also:

command	comment	unix equivalent
`delete system` *s_f1*	delete a single file	`rm` *file1*
copy-system*s_f1 s_f2*	copy a single file	`cp` *file1 file2*
`rename system` *s_f1 s_newname*	rename/move a single file	`mv` *file1 file2*
`set directory` *s_dirname*	change directory (cd)	`cd` *dirname*
`make directory` *s_dirname*	make a directory	`mkdir`
`Path` (directory)	returns the path to the current directory	`pwd`
`Sarray` (*s_filename_filter* `directory` [`all`])	returns the file list array, `all` goes to subdirectories	`ls` -1 [-R] *namepattern*

delete file

`delete system` *s_fileName s_fileName ...*

delete external file.

Example:

```
delete system "/tmp/aaa"
```

See also other internal icm equivalents of the system commands that allow to avoid new threads.

command	comment	unix equivalent
copy-system*s_f1 s_f2*	copy a single file	`cp` *file1 file2*
`rename system` *s_f1 s_newname*	rename/move a single file	`mv` *file1 file2*
`make directory` *s_dirname*	make a directory	`mkdir`
`set directory` *s_dirname*	change directory (cd)	`cd` *dirname*
`delete directory` *s_dirname*	delete an empty a directory	`rmdir`
`Path` (directory)	returns the path to the current directory	`pwd`
`Sarray` (*s_filename_filter* `directory` [`all`])	returns the file list array, `all` goes to subdirectories	`ls` -1 [-R] *namepattern*

delete history lines

`delete session`
deletes all previous history lines. Example:

```
call _macro
delete session
```

delete hydrogen

`delete hydrogen` *as*

`delete hydrogen` *chem* `[all]`
delete selected hydrogen atoms in a non-ICM object or a chemical array. See also `build hydrogen`. To delete hydrogens in an ICM object, `strip` it first.

When the hydrogens are deleted in a `chemical` array, the default is to preserve the **chiral** hydrogens in fused rings (the regular chiral hydrogens will still be deleted). To delete all hydrogens use option `all` . In the latter case when the hydrogen carrying the stereo bond is deleted for all heavy atoms including fused rings and the stereo bond will be reassigned to one of the heavy atom neighbors.

Example:

```
build string "ASD"
strip # makes a non-ICM object
delete hydrogen a_/2,3/h*
#
group table t Chemical("[C@@](C)(N)[H])O") "mol"
delete hydrogen t.mol all # all hydrogens gone
```

delete object

`delete` { `object` | *os* }
delete molecular object. Make sure that you specify an object selection (`a_1crn.` is correct, `a_1crn.*` or `a_1crn.//*` is INCORRECT.) To delete an object from a selection variable (`as_out`,as2_out or `as_graph`, or any use defined `aselection` variable), use `delete atom` *as_namedSelection* (e.g. `delete atom as_graph`) or specify the selection level explicitly.
Examples:

```
   delete object           # delete ALL molecular objects
   delete a_*.             # delete ALL molecular objects
   delete a_2,4.           # delete objects number 2 and 4
   delete a_2a*.           # delete objects with names starting from 2a

   read pdb "1crn"         # load crambin
   convert                 # create the second object named 1crn_icm
                           # from the pdb object
   delete a_1.             # delete the 1st pdb-object
   delete Object( as_graph )  # graphical selection
```

delete molecule

`delete` [`molecule`] *ms*
delete separate molecules from molecular objects. The integer reference number(s) of molecule(s) which can be shown by the `show molecule` command and used in molecule selections are redefined after deleting or `moving` molecules from or in the ICM-tree, respectively.
To delete a molecule from a selection variable (as_out,as2_out or as_graph, or any use defined `aselection` variable), use `delete atoms` *as_namedSelection* (e.g. `delete atom as_graph`) for non-ICM objects, or use the `Mol` function to specify the selection level explicitly (e.g. `Mol( as_graph )`).
Examples:

```
   read pdb "2ins"         # load insulin with water molecules
   delete a_2ins.w*        # delete water molecules
   delete atoms as_graph   # deletes selected non-ICM atoms/molecules
   delete Mol( as_graph )  # deletes selected non-ICM atoms/molecules
```

delete bond

`delete bond` *as_singleAtom1 as_singleAtom2*
delete a covalent bond between two selected atoms. This command is used to correct erroneous connectivity guessed by the `read pdb` command. It is particularly important when you are going to create a new ICM-residue using the `write library` command and the entry to it in the `icm.res` or your own residue file (it has the same format). In interactive graphics mode you may type `delete bond` and then click two atoms with the CTRL button pressed.
Examples:

```
   read pdb "newmol"              # automatic bond determination is not perfect
   delete bond a_/3/cg1 a_/5/ce2  # disconnect two carbon atoms
```

See also: `make bond` and `make bond` *atom_chain* .

delete boundary

`delete boundary`
an auxiliary command to free additional memory allocated by the `make boundary` command.

delete conf

`delete conf` *i_stackConfNumber* [*os_obj*]

`delete conf` *i_confNumberFrom i_confNumberTo* [*os_obj*]

`delete conf` *I_stackConfNumbers* [*os_obj*]
delete a specified conformation from the stack or a series of conformations starting from *i_stackConfNumber* to *i_stackConfNumberTo* . An integer array of indices can also be provided.

if the *os_obj* argument is provided the changes above will be applied to the local stack in the object.

delete drestraint

`delete drestraint` [*as_1* [*as_2*]]
delete distance restraints formed between specified atom selections *as_1* and *as_2*. If no selection is specified all distance restraints are deleted
Examples:

```
   delete drestraint a_mol1 a_mol2       # intermolecular restraints
```

delete label

`delete label` [*i_StringLabelNumber*]
delete graphics string label (text in the graphics window). These strings have no unique identification names, they are just numbered. Numbers are compressed as you delete some labels from the middle of the list.
Examples:

```
   delete label # deletes all labels
   delete label 1  # delete the first displayed label
```

See also:

`show label`	to find out the label number and
`display label`	to create and display a string label.

delete labels from 2D chemical spreadsheets

`delete label` *chemarray* `[all]` `[index=I_]`

deletes atom annotation in 2D chemical spreadsheet. Without the *all* option the command will only remove labels from the selected atoms, otherwise all labels will be removed. The selection can be done in the GUI and it appears as a green halo around select atoms.

Example:

```
# create annotated chem table
add column t Chemical({"CCCCN","CCCNCCC","CC(=O)O","C(=O)O"})
add column t Predict( t.mol "MolpKaBase" ) name="pkab"
add column t Predict( t.mol "MolpKaAcid" ) name="pkaa"
set label t.mol t.pkab window = {0.,14.}
set label t.mol t.pkaa window = {0.,14.}
# now delete it
delete label t.mol all
```

See also: `set label chemical`

delete link

`delete link` *ms*

delete links to sequences and alignments for selected molecules

`delete link variable`

delete **all** groups of linked variables (e.g. unlink the variables), see also `link variables` .

delete map

`*delete` { `*map` | `s_mapName` } delete *s_mapName* or all maps.

delete sequence

`delete sequence` [*seq_1 seq_2* ..]
`delete sequence` { `selection` | `compress` | `protein` | `peptide` | `nucleotide` | `unknown` | `swiss` }

- ♦ `selection` : delete the sequences *selected* through GUI.
- ♦ `compress` : delete the sequences not included in the alignments, i.e. freely floating sequence not included in any alignments, (compare with the `compress` option of `delete`)
- ♦ `protein` or `peptide` will delete only amino-acid sequences,
- ♦ `nucleotide` will delete only DNA or RNA sequences,
- ♦ `unknown` : delete sequences with more than 20% of 'X' or 'x' residues. Note that this option changed its meaning. Previously it was same as `compress` **.**

`delete sequence` *n_seq_at_the_end_of_seq_list*
`delete sequence` [*i_minLength i_maxLength*] `# delete OUTSIDE range.`

- ♦ no arguments: delete all ICM-sequences
- ♦ one integer argument: delete last *n* sequences from the sequence list
- ♦ two integer argument: delete sequences shorter than *i_minLength* or longer *i_maxLength*

Deleting some sequences from an alignment

`delete alignmentName only selection`

`delete alignmentName only` *seq1 seq2 ...*

To delete sequences *selected* via the graphics user interface from an alignment without deleting them from the shell. Example

```
  delete sh3 only Fyn
```

```
delete sh3 only selection
```

delete site

`delete site` *seq* [{*s_Site*|*i_number*|*I_numbers*|`pattern=`*s*}]

`delete site` *ms1* [{*s_Site*|*i_number*|*I_numbers*|`pattern=`*s*}]

`delete site` *rs*
delete the `sites` of the selected molecules. The sites can be specified by their name, or number, or residue selection. All sites are deleted by default.
Example:

```
nice "1as6"          # has 3 sites, one in each molecule.
delete site a_1.1 {1}
delete sites         # delete all of them
```

See also: `site`

delete sstructure

`delete sstructure` *seq_1 seq_2* .. `delete sstructure select`
delete the assigned secondary structure to prepare the sequence for the secondary structure prediction (see the `Sstructure` function).
The `selection` option allows one to delete secondary structure only for the sequences selected through GUI.

delete site in alignment

`delete site` *ali* [`i_number`] [`I_box`]

deletes annotation in the alignment by *i_number* or inside the *I_box*.

See also: `set site alignment`

delete disulfide bond

`delete disulfide bond` [`all`] [{ *rs_Cys1 rs_Cys2* | *as_atomSg1 as_atomSg2* }]
delete specified or all disulfide bridges in ICM objects.
Examples:

```
        # SS-bond specified by residue, or
delete disulfide bond a_/15 a_/29
        # by atoms
delete disulfide bond a_/15/sg a_/29/sg
        # remove all SS-bonds in the current object
delete disulfide bond all
```

See also: `make disulfide bond` and **(important!)** `disulfide bond`.

delete peptide bond

`delete peptide bond` [*as_N as_C*]
delete specified extra peptide bonds in ICM objects (e.g. imposed to form a cyclic peptide).
Example:

```
delete peptide bond a_/15/c a_/29/n
```

See also: `make peptide bond` and `peptide bond`.

delete stack

`delete stack`
delete the main `stack of conformations` in ICM shell. Be careful, there is a single share stack in the shell (deleted by this command) and each ICM object can also store a compressed stack of conformers.
See also `read stack`, `read stack`, `write stack`, and `delete conf`.

delete conformational stack inside an object

`delete stack` *os*

deletes the compressed stack inside the specified object.

See also:

- ♦ `store stack object`
- ♦ `load stack object`
- ♦ `montecarlo` .. `store`
- ♦ `set object` .. `stack`
- ♦ `Exist` (*os1* `stack`)

delete parray elements

`delete` *parray*[*i_index*]

`delete` *parray*[*I_index_list*]

deletes specified elements from a `parray`.

Example:

```
C = Chemical({"C","CC","CCC","CCCC","CCCCC"})
delete C[{1,3,5}]
delete C[1]
```

delete table

`delete` { *T_table* | *table_expression* }
delete the specified complete table or just the entries selected by the expression.
Examples:

```
   group table t {1 2 3} "a" {4. 5. 7.} "b"
   delete t.a == 2       # the second entry
   show t
   delete t[2]           # the second entry
   show t
   delete t              # the whole thing
   group table t {1 2 3} "a" {4. 5. 7.} "b"
   delete t.a > 1          # 2nd and 3rd
```

delete term

`delete term` *s_terms*
switch off the specified terms of the energy/penalty function.
Examples:

```
   delete terms "tz,sf"    # do not consider tethers and solvation contributions
```

delete selftether

`delete selftether` [*as*]

deletes internal tethers for selected (or all atoms)

See also:

- ♦ `selftether`
- ♦ `set selftether`
- ♦ `term ts`
- ♦ `convert`
- ♦ `set tether`

delete tether

`delete tether` [*as*]

delete tethers of the specified atoms (*as_*), if no selection is specified all tethers in the `current object` are deleted.

`delete tether loop` [*as*] - this tool deletes tethers for residues flanking insertions and deletions (one residue on each side), as well as N- and C- termini. The tool is used to help the `minimize tether` command to build a more relaxed loop or end.

---

delete a tree from a table header array

`delete variable` *treeParray i_treeIndex*

deletes a tree object (generically considered as a `parray`)

Example:

```
make tree T
delete variable T.cluster 1
```

delete selected chemical fragments

`delete chemical` *chemarray*

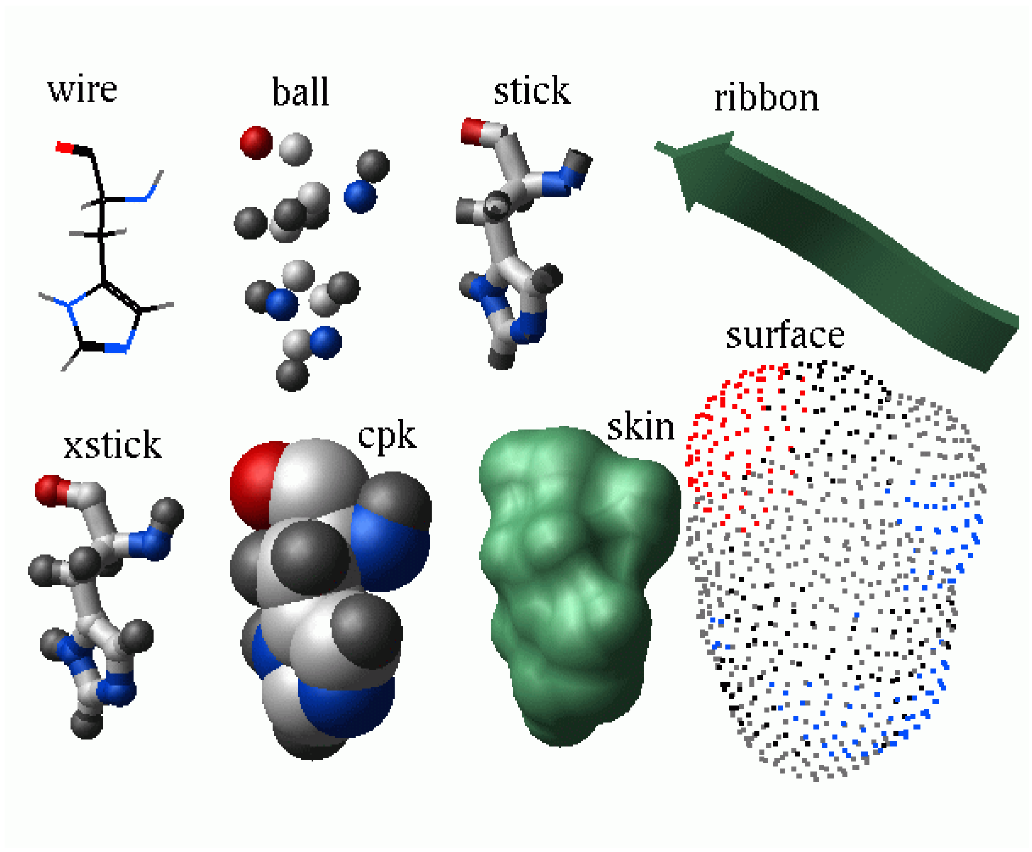
deletes selected parts of the chemicals. See `select chemical` command.

display

display molecules or graphical objects
display model

`display` [`wire`|`cpk`|`ball`|`stick`|`xstick`|`surface`|`skin`|`ribbon` [base]] [*as* [*as_2*]] [`color`] [`virtual`] [`center` [*center_options*]]

`display` [`transparent`] [`stick`|`skin`|`ribbon` [base]] [*as* [*as_2*]]
display specified graphics primitives for selected atoms or residues.



Once something is displayed and your cursor is in the graphics window you may rotate, translate, zoom and move both clipping planes with the mouse and keystrokes.
To refer to the base part of DNA/RNA represented as `ribbon` , use the additional specifier called **base**, which can be separately displayed and colored. E.g.

```
 makeDnaRna "ACTG" "mydna" yes yes "dna"
 display ribbon
 color ribbon base a_1 blue
```

Display surface atoms may be defined by TWO arbitrary selections (it would mean: display surface of atoms *as_1* as they are surrounded by atoms *as_2*) Note that the `GRAPHICS.hydrogenDisplay` preference may affect the displayed atoms. To be able to display all atoms set `GRAPHICS.hydrogenDisplay` to `"all"`.
Defaults: wire representation, all atoms (corrected by the GRAPHICS.hydrogenDisplay), coloring according to atom type.
color options
The color can be specified by a number of ways (see the `color` command for a more detailed description) : *Color* (e.g. `red`), s_Color (e.g. "red"), numerical color:
i_Color | *r_Color* | *I_Color* | *R_Color* [`window=` *R_2minmax*]
The window array of min and max values allows one to clamp the value you want to map to a color to the specified range.
Other options:
center : will perform the `center` command on the displayed object(s).
transparent : will display the ribbons, skins or sticks as transparent objects,

```
read pdb "1crn"
display transparent ribbon
display skin transparent
```

display surface refresh : will rebuild the surfaces with new `GRAPHICS.surfaceDotSize` values.
intensity= ***r_fraction*** **:** renders the image with fractional intensity by merging the source display image with the background.
virtual : additionally displays the coordinate axes, `virtual` atoms and virtual bonds starting

from the origin. It is a good way to visualize the whole ICM molecular tree as it grows from the origin. This option is applicable only to the ICM molecular objects.
More examples:

```
 build string "AFSGDH;QWRTEY"       # two peptides
 display                         # display current object and color atoms
                                 # according to atom type
 display a_1 red                 # display the first molecule and color it red
 display skin a_/5 a_* yellow    # display skin of the 5th residue
                                 # as surrounded by all the atoms
 display ribbon                  # display ribbon for all the residues

 read pdb "2drp"                 # a pdb file
 assign sstructure a_a/123:134,153:165 "H"       # No sstructure in 2drp
 assign sstructure a_a/109:114,117:121,141:144,147:151 "E"
 display a_a ribbon red                          # two Zn-fingers
 display a_a/113,116,143,146/!n,c,o xstick blue  # Cys residues
 display a_a/129,134,159,164/!n,c,o xstick navy  # His residues
 display a_m,m2 cpk magenta                       # Zn-atoms
 adna1=a_b//p,c3['],c4['],c5['],o3['],o5[']      # two DNA chains
 adna2=a_c//p,c3['],c4['],c5['],o3['],o5[']
 display adna1 xstick white
 display adna2 xstick aquamarine
 display adna1 adna1 surface white
 display adna2 adna2 surface aquamarine
 center
 display "Zn-finger peptides complexed with DNA" pink

# display 4 chains of insulin as 4 thick worms colored from N-to C-terminus
 read pdb "2ins"
 color background blue
 assign sstructure a_/* "_"  # thick worm representation
 GRAPHICS.wormRadius= 0.9
 display a_/* ribbon only
 color a_1/* Count(1 Nof(a_1/* )) ribbon
 color a_2/* Count(1 Nof(a_2/* )) ribbon
 color a_3/* Count(1 Nof(a_3/* )) ribbon
 color a_4/* Count(1 Nof(a_4/* )) ribbon

# examples of DNA and RNA ribbons
 nice "4tna"
 resLabelStyle = "A"
 display residue label
 color residue label a_/?u gold # ??u also selects modified Us
 color residue label a_/?a red
```

display new: refresh or unclip view

`display new`

`display restore`

`display restore plane`

commands to mimic some of the interactive controls. These commands are primarily used in GUI commands (see icm.gui file) and scripts/macros.
new : rebuilds some graphical representations (e.g. your as_graph has been changed in the shell and you need to refresh the image, or you changed the orientation and want to redisplay the labels elevated above the skin surface by `resLabelShift`).
restore : a softer action than `new` .
restore plane : moves the clipping planes beyond the displayed objects (keystroke: `Ctrl-U`, or the `'Unclip'` button) .

display off-screen

`display off` [*i_Width i_Height*]
Sometimes you want to generate some images in a script **without** opening an explicit graphics window. The `display off` command opens an off-screen rendering buffer of *i_Width* by *i_Height* size in pixels, in which all the usual `display/color/undisplay/center` commands work as usual. **NOTE:** one cannot have both off-screen and on-screen displays in one

ICM session.
An example script (can also be performed interactively):

```
   display off 400 300
   nice "1est"
   rotate view Rot( {0. 1. 1.} 50.)
   write image "est1"
   unix xv est1.tif
   set window 700 800  # NB: 'center all' will be applied
   write image "est2"
   unix xv est2.tif
   display a_/4/o cpk
   center a_/3,4
   write image "est3" rgb
   unix xv est3.rgb
   build string "se ala trp"
   display off 400 300
   display skin
   write image "est3" rgb delete
   unix xv est3.rgb
```

display origin

display the axis of the coordinate frame. The length of the arrows is defined by the `axisLength` parameter. Use `undisplay origin` to undisplay it. E.g.

```
 read pdb "1crn"
 display
 display origin
 undisplay origin
```

Setting rotation or rocking mode

`display rotate` [`on`|`off`] [*i_NofCycles*] [`pause`]

The graphics view can be set so that molecule is continuously rotating or rocking, but the ICM session remains interactive. This mode can be set with the above command. The style of continuous interruptable movement is controlled with the `GRAPHICS.rocking` preference. Specifying the number of rotation or rocking cycles *i_NofCycles* is useful for movie making. The `pause` option forces the command to finish the requested number of rotations before proceeding to the next commands, as opposed to just launching the rotation and proceeding with the rest of the script.

Example:

```
GRAPHICS.rocking = "xY-rocking"
display rotate on 3 # three cycles
```

See also: `write movie` , `GRAPHICS.rocking`

display stack

`display stack` [*os_withStoredStack*] [*iFrom iTo*] [`loop` [=*nCycles*]] [*r_NofInterpolationFrames* [`simple`|`cartesian`]] [`center`] [`sstructure`] [`auto`]

interpolated display of conformational `stack` of its parts. Aruments and options:

- ♦ optional *os_withStoredStack* . If this argument is missing, the global `stack` will be used. With the argument the built-in local object stack will be played out.
- ♦ optional start and end frames: *iFrom iTo*
- ♦ option `loop` [= *nCycles*] : the command makes a video loop and repeats it 99999 times. Optional *nCycles* redefines the number of repetitions.
- ♦ *r_NofInterpolationFrames* [`simple` | `cartesian`]] (e.g. `10.0 cartesian`) : determines the number of intermediate frames. The following interpolations are currently provided:
    - ◊ `simple` : just wait for the specified number of frames
    - ◊ `cartesian` : perform linear interpolation between stack conformations

  The default interpolation is `simple` .
- ♦ option `center` : centers on the displayed atoms

- option `sstructure` : recomputes secondary structure for each stack conformation.
- option `auto` : extracts the number of cycles (1 or endless loop) and the number of interpolated frames (no interpolation or a fixed number of interpolated frames) from the stack itself. The two parameters can be set with the `set stack` *os* `loop|fast [off]` command. The GUI interface for the object stack display uses the `auto` option.

Example:

```
  build string "ASDFW"
  montecarlo v_//x* mncalls=10 vwMethod=2 # create a conformational stack
  display xstick cpk  only
# the previous commands just prepare stack and display
  display stack 20. cartesian loop=4 center # repeat 4 times and stop
```

Another example in which the displayed trajectory is dumped into a movie file.

```
# make the same preparations
  write movie "peptamovie" on exact
  display stack 20. cartesian loop=1 center # repeat 4 times and stop
  write movie exit
```

The `display stack` command is somewhat similar to the `display trajectory` command. The `display stack` command has the following benefits:

- it recomputes the skin if the skin is present
- does not mess up the C-terminus in case of local deformations
- does not save or use any external files.
- it allows easy looping with the `loop` [= *nCycles*] option.

One relative disadvantage of the command is that only the `cartesian` interpolation is available, while `display trajectory` has other types of interpolations (e.g. cosine weighting, mixed cartesian/angular interpolation) .

See also:

- `display trajectory`
- `write movie`
- `store frame`
- `stack`

display box

`display box` [*R_6boxCorners*]
display graphics box specified by x,y,z coordinates of two opposite corners of a parallelepiped. This box can be resized and translated interactively with the Left and Middle mouse buttons:

- Resizing: Grab **a corner of the box with the Left-Mouse-Button** and drag it to resize the box
- Translating: Grab a corner or a center of the box with the **Middle-Mouse-Button** and translate

See also the `Box` () function which returns six parameters describing the box.
Examples:

```
   build string "se ala his gly met" # a peptide
   display
   display box                       # the default box
   display box {0. 0. 0. 2. 2. 2.}  # define position/size
   display box Box(a_/2 )            # surround the a_/2 by a box
   display box Box(a_/2 1.2)         # or add 1.2A margin
```

display clash

`display clash` [*as_1*] [*r_clashThreshold*]
display all the interatomic distances for selected atoms which are shorter than the sum of van der Waals radii multiplied by the *r_clashThreshold* parameter. The default value is taken from the `clashThreshold` variable. Initially it is set to `0.82` but can be redefined. IMPORTANT: this will work only for the ICM-objects. For hydrogen bonded atoms the threshold is additionally multiplied by `0.8.` Use the `show energy "vw"` command (and pay attention to the current fixation) to precalculate interaction lists.
This command may show some irrelevant short contacts. `calcEnergyStrain` , `display gradient` , etc. seem to be more informative.

See also: GRAPHICS.clashWidth , clashThreshold , show clash, undisplay and atom energy gradient (force) analysis with: show a_//G or display a_//G.
Example:

```
 read object s_icmhome+"crn"
 show energy "vw"
 display a_
 display clash               # all clashes, default clashThreshold=0.82
 undisplay clash
 display clash a_/11 0.95    # distances < (R1+R2)*0.95
# this is an alternative method which analyzes the gradient
 selectMinGrad = 100.        # analyzes forces greater than 100
 display ribbon grey
 display Res( a_//G )
 display gradient a_//G
 color Res( a_//G ) ribbon magenta
```

display drestraint

display drestraint *as*
displays drestraints, disulfide bonds, and peptide bonds imposed on selected atoms.
See also: read drestraint, set drestraint, make disulfide bond, make peptide bond, make drestraint.
Example:

```
  build string "se ala his trp ala gly gly"
  display
  set drestraint a_/1/hn a_def.a1/6/o 2
  show energy "cn"
  display drestraint
  minimize "vw,14,to,cn"
```

display gradient

display gradient *as*
display vectors of energy derivative with respect to atom positions or selected atoms as_ .
Important: the gradient must be **pre-calculated** by using one of the following commands: show energy or minimize . The values of gradient components (lengths of vectors for each atom) can be shown by show gradient *as_*. When a gradient vector is displayed, two transformations are performed: it is scaled and colored to represent the range of values in the most convenient and natural way while still being able to deal with a wide range of gradient values from negligible to 10 to the thirtieth power, as may be the case for a strong van der Waals clash. When all gradient vectors are under 20 kcal/mole*A they will be colored by the "cold" colors (blue...green...yellow) and will be assigned a length less than 2 Angstroms. If you see a red and long vector you may have a problem. Check it by zooming in and using show gradient *as_*. You can also select only atoms with gradient greater than the threshold value selectMinGrad by typing a_//G and display only specified strained atoms. It helps to get rid of little blue arrows for unstrained atoms.
Examples:

```
 build string "ala his trp glu leu"
 randomize v_//phi,psi
 show energy
 selectMinGrad= 20.
 display a_
 display gradient a_//G
```

display grob

display grob [solid][smooth][dot][reverse] [transparent]
display *g_Name1 g_Name2* ... options
display grob selection ... options
display all, specified, or graphically selected graphics object(s) . They are referred to as **grob** in the ICM-shell and as "3D meshes" in the GUI interface. The display grobs command will display all existing graphics objects. Options:

- ♦ dot will show only dot-vertices of the object.

- ♦ `reverse` to invert lighting; this option will change directions of the grob surface normals (will turn the grob inside-out)
- ♦ `smooth` enforces the Gouraud shading method to smooth the solid surface.
- ♦ `solid` allows solid surface representation of the object and requires that the original object has information about triangles forming the solid surface.
- ♦ `transparent` makes solid grob transparent

One can also `color` and `undisplay` graphics objects, as well as `connect` to them.
Examples:

```
   read matrix s_icmhome+"def.mat"      # 2D sin(r^2)/r^2 function of a grid
   make grob solid def                  # convert matrix into a graphics object g_def
   display g_def smooth                 # a hat of the 22st century
   rotate view Rot({1. 0. 0.}, 45.)     #
   display g_def reverse                # shine light from inside the head
   display grob smooth transparent      # like Lenin in Mausoleum
```

set font of a 3D label

`display` *g_label* [`bold`] [`italic`] [`underline`] *i_Size* [`font=`*s_FontName*] [`rgb=`*R_3rgb*|"#`xxyyzz`"]

displays *g_label* text (technically it is a `grob` with a single point and associated text) in a particular font.

Example:

```
read pdb "1crn"
display a_
label3d = Grob("label",Mean(Xyz(a_/3,4)), "3D label for res 3,4")
set font label3d times 36 rgb="#00ffdd"
display label3d
select edit label3d # makes it movable, press Esc to get rid of the cursor
```

display hbond

`display` hbond [*as*] [*r_maxHbondDistance*] [`only`]
Only `hydrogen bonds` of the `current object` may be displayed. Before calling this command, you should use any of the following commands: `show hbond`, `show energy`, `minimize` to calculate the list of hydrogen bonds. The real argument *r_maxHbondDistance* defines an upper bound of the distance between a hydrogen and a potential hydrogen acceptor to place the pair to the hydrogen bond list. (Default value of *r_maxHbondDistance* parameter is 2.5 A.) The list is recalculated for each new loaded molecular object. Hydrogen bonds on display are colored according to their hydrogen-acceptor distances. The option `only` allows one to display hydrogen bonds without corresponding molecular object. Longer and shorter H-X distances in the hydrogen bond are color-coded, from red to blue, respectively.
For ICM object the hydrogen bonds are calculated much faster because the atom pairs are precalculated. However, the displayed hydrogen bonds will then depend on how the model was fixed. No hydrogen bonds will be shown *inside rigid bodies*.

The color or hydrogen bonds will be calculated according to a calculation involving the effective lone pair density (see `hbond color`).

See also: `undisplay hbonds`, `show hbonds`.

Strength and color of a hydrogen bond

The hydrogen bonds created or displayed with the `make hbond` or `display hbond` commands are colored according to the estimated 'strength' of this hydrogen bond. This is just an estimate since the energy of hydrogen bond is not easily decoupled from the van der Waals and electrostatic contributions between the hbonded atoms and their immediate environment. In ICM the strength is estimated using the following procedure described in J Med Chem. 2003 Jul 3;46(14):3045-59.

For a hydrogen bond acceptor atom ***A(i)*** and a hydrogen atom ***H(j)*** located at $\boldsymbol{r_{j,}}$ the hydrogen bonding interaction was estimated

$\boldsymbol{F_{ang(phi)F_{dist}}(r_{LPi} - r_{j)}}$

, where ***phi*** is an angle formed by the hydrogen bond acceptor atom, hydrogen, and the hydrogen bond donor, and $\boldsymbol{r_{LPi}}$ is the radius vector of the center of the lone electron pair (LP) closest to the hydrogen. The angular function used was defined as $\boldsymbol{F_{ang} = 1 - cos(k*phi)}$. Parameter k is accessible as `GRAPHICS.hbondAngleSharpness` in the shell. Distance function $\boldsymbol{F_{dist}(r_{LPi-rj)}}$ was constant (1.0) within $\boldsymbol{L_{HB/2}}$ from the lone pair center and dropped as

exp($\boldsymbol{-(((r_{LPi} - r_{j)/L_{HB)}} - 0.5)^2}$) .

beyond that distance, where $\boldsymbol{L_{HB}}$ is the characteristic range of hydrogen bonding interaction (value of L=1.6 ï¿½was used). Lone pair centers were placed at 1 ï¿½from the hydrogen bond acceptor atom, assuming symmetrical planar trigonal configuration for sp2 atoms and tetrahedral configuration for sp3 atoms. The resulting functional dependence reflects (at least qualitatively) the physical nature and observed statistics of the hydrogen bond interactions. The interaction is maximized when the hydrogen atom is pointing directly to the acceptor atom along a lone pair axis and drops quickly as the hydrogen is moved farther away. The strength declines more gradually as the hydrogen moves out of the LP axis or, as hydrogen bond donor, hydrogen atom, or hydrogen bond acceptor, move out of alignment.

See also: `GRAPHICS.hbondMinStrength`

display label

`display` [{ `atom` | `residue` }] `label` [`selection`]
`display variable label v_selection`
a graphics label with atom name, residue name, variable name for all or selected atoms, residues or variables respectively. The text of this label is not user-defined, although you can control it in two different ways. First, residue label style can be set using either `Ctrl-L` in the graphics window or `resLabelStyle` preference , and variable label style either by `Ctrl-V`, or setting `varLabelStyle` preference. Second, the ICM-shell string variable `s_labelHeader` defines a prefix string for all labels. For example, if you display CPK atoms you may move the label to the right from the atom center by `s_labelHeader="  "` .
The `_aliases` file has convenient aliases (e.g. `ds` for display, `unds` for undisplay, `re` , for residue, `va` for variable) for those of us who like typing commands. In this case you may just type `ds va la` to display variable labels, etc.
Examples:

```
 build string "FAHSGDH"
 display a_
 display residue label #
 undisplay label
 display residue label a_/his
 display variable label v_//phi,psi
 display variable label v_//* & as_graph
 display atom label a_/1:3/*
 undisplay label
# or with aliases:
 ds re la a_/1,3
 unds la
 .. etc.
```

display map

`display` { `map` | *map_name* } [*I_colorTransferFunction*] [*R_2RangeOfMapValues*]
displays a real function defined on a three-dimensional grid (i.e., an electron density `map`). Optional `iarray` argument defines a color transfer function according to deviation from the mean.
If you provide an explicit range of map values (*R_2RangeOfMapValues*), the map values will be clamped into this range, divided into `Nof(` *I_colorTransferFunction*) subranges, and colored according to the values of *I_colorTransferFunction* :

- 0 - transparent/invisible
- 1 - blue
- maxNumer - red

To undisplay the bounding box reset the `GRAPHICS.displayMapBox` parameter.
See also the `color map` command.
Example:

```
   build string "se his arg"
   make map potential "el"  Box( a_/1,2/* , 3. )
   display a_
   display map m_el {1 2 0 0 0 0 3 4 5 6} {-20.,100.}
   center
   make grob m_el 2. name="g_1"
   make grob -m_el 1. name= "g_2"
   display g_1 red
   display g_2 blue
```

In the `display map m_el {0 1 2 3 4 0} {-2.,2.}` example, the values will be clamped into the -2.,2. range. The range will be divided into 6 sub-ranges: `-infnty:-2., -2.:-1, -1:0, 0:1, 1:2, 2:+infnty` . The first and the last ranges will be invisible (color 0). The four ranges in the middle will be colored from blue to red.

See also related commands: `read map`, `write map`, `delete map`, `show map`, `set map`, `make` (1), `make` (2) and file format `icm.map` .

display trajectory : simulation trajectory

`display trajectory` [*s_TrjFileName*] [*i_From* [*i_To*]] [*r_Smooth1* [*r_Smooth2*]] [*as_1*] [`center` [*as_2*]] [`sstructure`] [*imageOptions*]
lets you play, stop and reverse a Monte Carlo simulation trajectory as well as write a series of images for future assembly of those images into movies.

Arguments and options:

- ♦ Integers *i_From* and *i_To* specify the frame range.
- ♦ Real values *r_Smooth1* and *r_Smooth2* determine minimum and maximum smoothing parameters (i.e. number of additional frames, inserted if conformation change is too dramatic). For example: `100. 700.`
- ♦ Specifying atom selection *as_1* defines a certain fragment on to the initial conformation, of which subsequent conformations are superimposed.
- ♦ The *image* saving *options* include: `image` [=*s_framePath*] [`rgb`|`targa`|`png`|`gif`] Option `image` allows one to automatically save a series of image files in the *s_framePath* argument of the `image=` option or in the default `s_tempDir` directory.
- ♦ `center` option with selection *as_2* determines a fragment for graphics window centering (all, if `center` without *as_2*).

To obtain the trajectory info use
`read trajectory` *s_TrjFileName*
When playing a trajectory, you can use ICM interrupt (`Ctrl-\`) to stop, and then toggle stepwise frame playing, reverse, or quit playing. The default is to play a whole trajectory without smoothing, superimposition or centering. Example:

```
build string "ala ser ala thr ala glu ala"
mncallsMC=10000                         #
montecarlo trajectory
read trajectory "def"
ds ribbon, wire
ds trajectory center sstructure  10.
```

Notes:

- ♦ do not forget to start ICM with the **`-24`** `flag` to double the image quality.
- ♦ set `IMAGE.generateAlpha` to `no` if you want to *keep* the background colored and not transparent.

Allowed image formats are: `rgb, targa, png, gif` . The file extensions will correspond to the image file format. The image file names consist of the default path and name, appended with the frame number. Example:

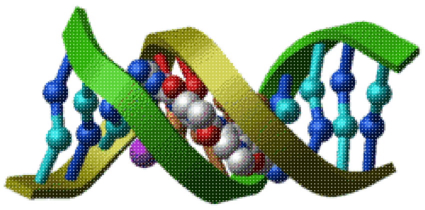
```
display trajectory image="/tmp/f"
/tmp/f_1.png
/tmp/f_2.png
```

```
...
s_tempDir = "/home/jack/X"
display trajectory image rgb
/home/jack/X_1.rgb
/home/jack/X_2.rgb
...
```

All the other image preferences may be predefined by the `IMAGE` table.
Option `sstructure` will dynamically reassign secondary structure while going through conformations of each frames. This option is very useful if you perform peptide/protein simulation and want to see if secondary structure elements are forming transiently.
See also: `trajectory file`.

display ribbon

`display ribbon` *rs color*



displays protein or DNA backbones in `ribbon` presentation.

See also:

- ♦ `ribbon`
- ♦ `undisplay`
- ♦ `ribbonStyle`
- ♦ `GRAPHICS.ribbonCylinderRadius`

display site

`display site` *rs* `color`
display `site` information. Switch between different types of the site information with the `SITE.labelStyle` preference. By default only non-zero priority sites are displayed.

display skin or dotted surface

`display` { `skin` | `surface` } *as_1 as_2*

`display skin` *as_1* `molecule`
display analytical molecular surface, also referred to as `skin`, or solvent accessible `surface area` . Each `display skin` command will delete the previously displayed skin in the current `plane`. To display several different skins, use the `set plane` command to change the current graphics plane before you issue the display skin command. You can also convert the skin into a `grob` with the `make grob skin` command. You can co-display many grobs on the same plane, as well as make the grob transparent. This grob can be further split into individual shells with the `split` command.

Options:

- ♦ `molecule` : (for skin) considers each molecule in isolation

See also: `How to display and characterize protein cavities.`
Example:

```
build string "se ala his glu"  # test tripeptide
display                        # the wire model
display skin a_/1 a_/1         # skin around the 1st residue or just press <F1>
display skin a_1 molecule      # equivalent to a_1 a_1
set plane move on 2            # key with your cursor in the graphics window
```

```
   display skin a_/3 a_/3         # skin around the 3st residue
                                  # now you can toggle planes with F2 and F1
   display surface                # solvent-accessible surface
```

display slide

`display slide` [`reverse` | *i_slide*] [*s_slideProperties*] [`view`] [`smooth`] [`add`]]

display the next slide or slide number *i_slide* . Options:

- ♦ which slide to show? if you have just said: `display slide` the next slide will be shown, `display slide reverse` will show the previous one. A specific slide number *i_slide* can also be shown (ICM also understands `index=3`).
- ♦ `view` : using only the viewpoint/clipping planes from a slide (see also `set view`).
- ♦ `smooth` : or `smooth=` *i_transition_time_in_msec* will make a smooth view transition from the current state to the slide view. (e.g. `display slide smooth` or `display slide 5 smooth=1000`
- ♦ `add` : adding representations to the existing display, rather than overwriting the slide (like appending a new graphical layer)

You can **individuall** control which sections of the slide information to use in `display slide` using *s_slideProperties* . The syntax of this string is the following: `"`*`sect1on`*`;`*`sect2on`*`;..;-`*`sect3off`*`;`*`sect4off`*`; .."` The section names are separated by a semicolon, and plus and minus are used to switch things on and off with respect to the default state. The allowed sections include:

Section Name	Default	Description
`"layout"`	-	if +, sets the layout of ICM windows and panels (if off, preserves the current layout)
`"activewindows"`	+	if +, sets the saved active window or panel in ICM gui
`"smooth"`	-	if +, makes smooth animated view transitions between slides
`"add"`	-	if +, adds the next slide as a layer to the previous, rather than overwrites it
`"gf"`	+	graphical representations (CPK, xstick, skin etc.)
`"color"`	+	colors of representations
`"labeloffs"`	+	restoring slide-specific displacements of residue labels
`"viewpoint"`	+	the view point, zoom, and clipping planes
`"graphopt"`	+	the state and parameters of rotation, rocking, etc.
`"mol"`	+	if - , do not restore any property of molecular objects in main graphics window
`"grob"`	+	if - , do not restore any property of grobs in main graphics window
`"map"`	+	if - , do not restore any property of maps in main graphics window
`"all"`	-	switches all sections, on (+) or off (-)

Examples:

```
 display slide 4 "-all;+gf;+color"
 display slide 4 "-viewpoint"
 display slide 4 "+smooth" # enforce smooth view transitions
```

`display slide show` [`index=`*i_start*] [`reverse`]

the keyword `show` switches the program into the slideshow mode and makes `smooth` transition the default. Other options are the same as above.

Examples :

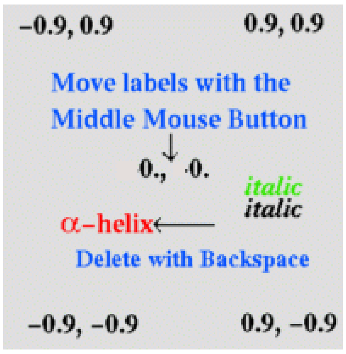
```
icm -g&
read binary s_icmhome+"example_slideSGC.icb"
display slide
display slide # the next slide
display slide smooth # make a 500msec-transition
display slide 4 # 5th slide
display slide 2 view  # enforce viewpoint from slide 7
display slide add 2 # display additional representations from slide 3
```

See also: `add slide`, `set view`.

display string

`display string` *s_StringText* [*P_image*] [`size=`*r_imageScale*] [*color_spec*] [*font_spec*] [*r_XscreenPosition r_YscreenPosition*]

display a text string in the graphics window. Relative X and Y screen coordinates (ranging from -1. to 1.) of the string beginning may be specified to display the string in a given location. Defaults are x = -0.9, y = 0.9, i.e. upper left corner of the screen.



The string can be dragged later to any location by the middle mouse button.

The command supports `various formats for specifying the label color` *color_spec* and `font parameters` to characterize the label font *font_spec.*

Two fonts are at your disposal: the default font (usually times) and the auxiliary font (usually symbol). Both fonts can be redefined by the `set font` command. You can also switch to the auxiliary font and back inside the string by backslash-A (**\A).** (.e.g "Red: \Aa\A-helix"). You can also list and delete your string labels by the `list label` and `delete label` commands. Examples:

```
display string "Crambin"                            # a simple string

display string "Act.site of \Ab\A-lactamase" yellow # Greek beta letter

build string "ala"
display string Name(a_1.) red 28, 0. 0.9     # first object name
                                             # in the middle
                                             # (font size=28)
```

display tethers

`display tethers` [*as*] [*r_minDeviation*]
displays tethers assigned to the selected atoms `as_` with deviation larger than *r_minDeviation.* Tethers can be imposed between atoms of an ICM-object and atoms belonging to another object, which is static and may be a non-ICM-object. (0. by default).

display volume

`display volume`

activates fog from the command line. See also `fogStart` . Accordingly, the `undisplay volume` switches the effect off.

display window

`display window` [*i_xLeft i_yDown i_xSize i_ySize*]

`undisplay window`

displays/undisplays the graphics window. When ICM is started without GUI, it is allowed to specify the window size and position.

See also: `set window`

`display` `window=`*s_windowList*

`undisplay` `window=`*s_windowList*

Displays/undisplays GUI windows and toolbars. *s_windowList* should be a comma-separated list with ICM panel and toolbar names.

- `"opengl"` # undisplays the graphics window
- `"all"` # undisplays all except graphics window
- `"alignments"`
- `"htmls"`
- `"masterview"` # shows/hides workspace panel
- `"moledit"` # shows/hides molecular editor window
- `"plotdialog"` # shows plot dialog for the current table (in modal mode)
- `"searchwindow"` # show/hides chemical search space window. Chemical pattern can be provided optionally as an extra argument
- `"columnfilter"` # launches column filter dialog for specified table column
- `"tablesearch"` # launches table "Find and Replace" for the active table. Search string can be provided optionally as an extra argument
- `"processes"` # shows/hides background job list window
- `"prop"` # 'Display Panel' with multiple tabs (display/light,...)
- `"terminal"`
- `"tables"`

Also the tool panels:

- `"moveTools"`
- `"clipTools"`
- `"miscTools"`
- `"viewTools"`
- `"planeTools"`
- `"fileTools"`
- `"levelTools"`
- `"tableTools"`

You can also display/undisplay individual tabs from the 'Display Panel'. To do that you need to append a tab name to the "prop:".

Example:

```
undisplay window="prop:light"   # hide 'light' tab from the panel
```

Note: This command does not affect the content of the main working area (the center)

Example:

```
read binary s_icmhome + "example_search.icb"
display a_
undisplay window="tables" # hides tables
undisplay window="all"    # leaves only 3D graphics window
display window="moledit" Chemical("CCO")  # popups Molecular Editor with compound
display window="searchwindow" Chemical("CC[O;D1]")  # popups Chemical Search Space with
```

`display` `window=`*s_window* `center`

sets the specified *s_window* to the center. Windows which may occupy the central position are:

- `"opengl"`
- `"alignments"`
- `"htmls"`
- `"tables"`

Example:

```
read binary s_icmhome + "example_search.icb"
display a_
display window="opengl" center # sets graphics window to the center
```

display window=*s_layoutString*

Applies the window layout specified in the *s_layoutString*. ICM stores the layout information as a string in a specific format. Window layout information is stored, for example, in slides.

Example:

```
read binary s_icmhome + "example_search.icb"
sl = Slide()
display a_
display window=String(sl gui) #changes the view back to what was before the 'display' c
```

See also: Slide, String slide gui

---

display GUI windows

display gui [off] s_window

Obsolete command. See: display window

---

edit

edit icmShellVariable
interactively edit the ICM-shell variable using your favorite editor defined by the s_editor variable.
Examples:

```
   edit mncalls    # actually it is easier to type: mncalls=333
   edit FILTER     # edit a system table, do not change names of components
#
  group table t {1,2,3} "A" {"a","b","c"} "B"   # create a table
  t                                             #
  edit t                                        # edit table t
```

---

elseif

elseif
is one of the ICM flow control statements, used to realize conditional statements.
See also: if, then, and endif .

---

endfor

endfor
is one of the ICM flow control statements, used to perform a loop in ICM-shell calculations. See also for .

---

endif

endif
is one of the ICM flow control statements, used to realize conditional statements.
See also if, elseif, and then .

---

endmacro

A command ending a macro .
Examples:

```
   macro threeEssentialsOfLife          # declare new macro
                                        # define essentials
      l_info=no
      modes={"\n\tOoops!!\n","\n\tOuch!!\n","\n\tWow!!\n"}
```

```
                                        # randomly pick a line
      print modes[Random(1,3)]
   endmacro
   threeEssentialsOfLife                # invoke macro
```

Enumeration of stereoisomers

`enumerate chiral` *chem_array* `[index=I_selectedChems]` `[center=i_Max_Number_of_Centers]` `[name=s]`

Generates all possible stereo isomers for each chemical compound from or from selected chemicals (*I_selectedChems*). **Important:** this operation requires that two conditions are satisfied:

- ♦ a molecule has a stereo center (i.e. an sp3 atom with four **different** substituents
- ♦ if a stereo center has a definite chirality (`"up"` or `"down"`, or R, or S) stereo isomers will **not** be generated. The center needs a stereo bond is marked by type `"off"`, or "either" to imply an uncertain chirality or a racemic mixture of two isomers.

Sometimes you may want to skip compounds with number of unspecified centers greater than certain value. In this case you should provide `center` = *i_Max_Number_of_Centers* argument to the command.

The command will always generate at least one element for each compound. Example:

```
group table t {"CC(N)O","CC(C)C(C)O"} "mol"
enumerate chiral t.mol name="isomers" # creates isomers.mol
```

Tautomer enumeration

`enumerate tautomer` *chem_array* `[keep]` `[filter]` `[index=I_index_array]` `[name='T_tauto']`
Generates all possible tautomers for each chemical compound from . Returns the resulting chemical array of tautomers. The command will always generate at least one element for each compound.

The current function only generates tautomers that preserve the atom content (does not add or remove hydrogens). With *keep* option it'll also preserve the hybridization state of each atom (i.e. does not change sp3 to sp2).

Some tautomers are formally possible but chemically do not make much sense. To avoid generating those tautomers, Split uses the TAUTOFILTER.tab file that contains the unwanted or chemically impossible tautomer patterns in the SMARTS format. Feel free to add more patterns to this file. Use *filter* option to enable filtering by patterns.

Example file:

```
#>T TAUTOFILTER
#>-sm----------------comment
"*C([OH1])=[N;R0]"  "peptide bond"

 p = Chemical( "C(=C(NC(=N1)N)N2)(C1=O)N=C2" )
 Nof(p) # 1 element
 1
 enumerate tautomer p
 show T_tauto
#>T T_tauto
#>-mol---------idx--------
  "C1=NC2=C(NC(=N)NC2=N1)O" 0
  "c1nc2=C(NC(=N)N=c2[nH]1)O" 0
  "C1=NC2=C(N=C(N)NC2=N1)O" 0
  "c1nc2c(nc(N)nc2[nH]1)O" 0
  "C1=NC2=C(NC(N)=NC2=N1)O" 0
  "c1nc2C(NC(=N)Nc2[nH]1)=O" 0
  "c1nc2C(N=C(N)Nc2[nH]1)=O" 0
  "c1nc2C(NC(N)=Nc2[nH]1)=O" 0
  "c1nc2C(=NC(=N)Nc2[nH]1)O" 0
```

Combinatorial library enumeration

`enumerate library` [`simple`] *chem_scaffold_R1R2 .. chem_R1 chem_R2* .. \ [`name=` *s_libTableName*|`output=` *s_fileName*] [`filter=`*expression*]

Applies chemical arrays (usually a column in a chemical table) for each of replacement groups *chem_R1*, *chem_R2* etc. to the first element of the scaffold template array. (also known as `enumerate library`

The scaffold. The scaffold structure needs to be drawn as a Markush structure, e.g.

```
add column scaffld Parray( "[R2]C(C(=O)[R3])NC(=O)N[R1]" ) name="mol"
```

The replacement groups. Each replacement group in a chemical array (table) needs to have an attachment point specified. In the Smiles/Smart representation used in ICM it is marked by an asterisk (e.g. `"[C*]CC"`). Marking an atom as an attachment point can also be done in the Chemical Editor (right-click on an atom and choose the Attachment Point menu item).

The output table or file. The output table will contain all combinations . If the `output` option is specified the resulting library is saved to a file and the table is not created. Warning: if the number of combination exceeds `20000` the resulting library is saved to a file automatically (to avoid memory problems).

- ♦ Option `name` = *s_table* allows one to change the default name of the output table.
- ♦ Option `output` = *s_file* forces the file output and suppressed the table creation.

The output chemical table has a product column as well as the index of each R-group.

simple option toggle a special mode where instead of full enumeration it simply goes through the input substituents and take i-th element from each. This mode requires that size of all R-group arrays should be the same. The size of the output will be equal to the size of R-group array(s)

Dynamic filtering of the output by applying a `filter` expression. The `filter=` *s_expression* option allows one to apply a filter during the library generation. The filter expression is a double-quoted string with the following structure: "*Function1 relation value* **&** or **|** *Function2 relation value* **&** or **|** .. "

Example:

```
filter = "MolLogP<5. & Nof_Frags('C(O)=O')<1"
```

The list of functions is expanding. The current list of the functions is the following:

Function Name	Description	Example
`MolWeight`		MolWeight < 650
`Nof_Molecules`	the number of individual molecules,including ions and salts	Nof_Molecules==1
`Nof_Chirals`	the total number of racemic and chiral centers	Nof_Chirals==0
`Nof_RotB`	rotatable bonds	
`Nof_HBA`	hydrogen bonding acceptors	
`Nof_HBD`	hydrogen bonding donors	
`Nof_Atoms`	the total number of non-hydrogen atoms	
`Nof_Frags` (*s_smart*)	counts the number of fragments	Nof_Frags('[S,P](=O)=O')==1
`DrugLikeness`	a number around 0	DrugLikeness > 0
`MolLogP`	log P prediction	
`Volume`	3D molecule volume prediction	
`MolPSA`	polar surface area	
`MoldHf`	heats of formation	
`MolLogS`	solubility	

See also:

- ♦ `Predict` for a detailed description of some of the functions.
- ♦ `make reaction`

A short form of the enumerate library command and linking. The replacement group arrays can be linked to the R positions of a scaffold with the `link group` command. In this case a short form of the command can be used, e.g.

```
link group scaffold.mol 1 r1.mol 2 r2.mol 3 r3.mol
enumerate library scaffold.mol
```

Example:

```
 read binary "example_enum.icb" # contains scaffold and R1,R2,R3
 enumerate library scaffold.mol[1] name=Name( "lib", unique ) R1.mol,R2.mol,R3.mol
 split group scaffold.mol[1] lib.mol   # if you want to split it back
```

The inverse operation: split the library into scaffold and replacement group arrays.A library can be also reduced back to the scaffold and replacement groups using the `split group` *scaffold library* command. E.g. `split group scaffld.mol combilib.mol`

See also: `make reaction` , `split group` , `Replace chemical` .

endwhile

`endwhile`
is one of the `ICM flow control` statements, used to perform a `loop` in ICM-shell calculations. See also `while` .

exit

`exit` [*s_message*]
exit from a `script` file to interactive mode. Do not confuse this command with the `exit` option in, say, `highEnergyAction` preference.
Similar to `return` [`error` *s_message*] from a `macro` .
To quit the program, use the `quit` command.

find

a family of commands for sequence and pattern searches, chemical matching, 3D pharmacophore matching, and alignment optimization. For chemical matching also see `chemical tables`, and the `Nof chemical` function.

find alignment : automated structural alignment

`find` *ali_initial* [`superimpose`] [*r_threshold*= `3.` [*r_retainRatio*= `0.5`]]
find the best structural alignment of two proteins by refining the inaccurate initial alignment *ali_initial* with the goal of finding the largest possible subset of residues which have similar local backbone fold in 3D space.
Option `superimpose` automatically superimposes molecules according to the found structural alignment upon completion of the iterations. This command needs a starting alignment of 2 sequences `linked` to the molecules with at least one atom per residue. If `Ca` atoms are not found the atoms carrying the residue label (see the `set label` command) are used.
Low gap penalties of 1.8 and 0.1 are recommended for the initial sequence alignment.
Algorithm : At each step aligned pairs of atoms which are further than *r_threshold* from each other are disconnected so that at least *r_retainRatio* pairs are be retained. Then the molecules are superimposed again and new residue pairs are tested and accepted if it leads to a lower overall rmsd. Warning: the result strongly depends on the relevance of the starting alignment to the best 3D alignment. Sometimes 3D irrelevant sequence alignment pairs do not tend to disconnect to allow transformation into a global 3D alignment: e.g. if only one pair of elongated helixes is aligned in the starting alignment and it is only a small part of an optimal alignment which would be completely different, it might not be eventually found.

See also other types of structural searches and superpositions:

- ♦ `find pdb`: search a database of a single structure for a fragment with a given sequence pattern and partial structural similarity (e.g. loop ends match).
- ♦ `superimpose`: performs structural superposition, the command can do it on the basis of sequence alignment on the fly.

Example:

```
read pdb "1nfp"
read pdb "1brl.b/"
rm !Mol(a_*./A )
make sequences a_*.
aa=Align(1brl_1_b 1nfp_a)
ds a_1.//ca,c,n grey
ds a_2.//ca,c,n green
superimpose a_1.1 a_2.1 aa
center
find aa superimpose
show aa

gapExtension = 0.05
ab=Align(1brl_1_b 1nfp_a)
find ab 4. 0.7 superimpose
show ab                          # better
```

See also: `pairwise alignment` `multiple alignment`

find database: sequence and pattern searches

`find database` [*r_probabilityThreshold*] `options`

`find database exact` [`distance=` *i_nOfMutations*]] `options`

`find database pattern=`{ *s_pattern* | *S_patterns* } `options`

`find database write` [*s_database*]

`find database fast` [= *i_speed*] [`output=`*s_file*] [`name=`*s_tabName*] # blast like fast search.
fast sequence or `pattern` search through a sequence database.
The default `find database` sequence search program performs a full gapped optimal sequence alignment, which is a global alignment with zero-end-gap penalties (ZEGA). These alignments are more rigorous (not heuristic) than popular BLAST of FASTA searches. The latest statistics of structural significance of sequence alignments derived for a number of residue substitution matrices will be applied (`Abagyan and Batalov, 1997`) to assess the probability that a matching fragment shares the same 3D fold. The *r_probabilityThreshold* (default `0.00001` or 5.) option defines the lowest acceptable probability of hit. You can also provide a `-logP` number (e.g. `5.5`) instead of a small probability (`10^-5`). Threshold of 10/DatabaseSize is usually a safe threshold (no guarantees though). Practically 10^{-5} is a safe threshold for a SWISSPROT search (65,000 sequences). At 10^{-4} you may find interesting hits, but a more serious analysis may be required to confirm its significance.
The second version of the command with the `exact` keyword performs a very fast search for identical or almost identical sequences. The `distance=` *i_maxNofMutations* parameter specifies the allowed number of mutations.
find database pattern

The third version of the command searches for string patterns in a sequence database. The `sequence patterns` can contain while cards (e.g. "A?[LIV]?\{3,5\}[!P]"). This search is very fast.
The fourth command `find database write` is used to export ALL sequences from the blast-formatted files into to an external FASTA file defined by `output=` string (default *s_databasePath.seq*). This option is the inverse of the `write index sequence` command which creates several BLAST files from a FASTA file.
The common options are as follows: [*s_databasePath* ("`pdbseq`")] [*seq_1* ..] [*ali_1* ..] [`output=` *s_outputFileNameRoot*] [`name=` *s_tableName*] [`unique`] [`delete`] [`protein` | `nucleotide` | `type`]

- ♦ DATABASE: *s_databasePath* (default: "pdbseq" files in the $BLASTDB directory) defines the path of the three files with the compressed sequence files. For compatibility these three files (.bsq, .atb, .ahd) are the same as generated by the `setdb` (BLAST) command. The available files can be vied with the `list database` command, by default the "swiss" file is taken from the $BLASTDB directory. If the environment variable $BLASTDB is set, the three files will be taken from this directory. To read database files from any directory, specify its explicit path (e.g. "./myLocalDb" or "/home/user/myHomeDb1") **Note:** when the PDB sequences are updated, the blast files go into `s_userDir` + "/blastdb" , On Linux the database is at "~/.icm/blastdb/pdbseq" . To make this directory the default blast directory, reset the `s_blastdbDir` to

"/*your_home*/.icm/blastdb/". In GUI, choose **File;Preferences;Directories** and modify the s_blastdbDir variable.

- QUERY: *seq_1* .. (list of sequences), or *ali_1* .. (list of alignments), or keyword selection determines which sequences will be searched against the database. The default (no argument) means that all the sequences currently present in the ICM-shell (see list sequence) will be searched. The selection can be made from the ICM GUI.
- OUTPUT FILES: option output= *s_projName* to redefine the name of the project. The default name of the output files is the name root of the database file. The following files are saved

```
 projName_seq  # query sequence(s)
 projName.seq  # a sorted list of database sequences truncated to the matching f
 projName.tab  # the result table
```

- TABLE: option name= *s_resultTableName* defines the names of output table which is created after the search. The table contains the boundaries of the hits, sequence identities etc.
- option margin= *i_seqMargin* in the pattern search defines the length of flanking sequences added to the matching fragment and saved in the *s_projectName.seq* file for further retrieval. Specify a very large number to store complete sequences.
- option delete will overwrite the output files without asking, as if l_confirm=no .
- option unique makes the program ignore hits with sequences 100% identical to the query set (if one sequence is a fragment of another, they are is still considered 100% identical).
- option protein or nucleotide limits the search to database sequences only of this type. It is important for PDB sequence database since it contains both protein and nucleic acid sequences.
- option type automatically selects protein or nucleotide based on the *query* sequence type, but only if you search with a *single* sequence.

Other important variables:

- alignMinCoverage (default 0.5) a threshold for the ratio of the aligned residues to the shorter sequence length.
- alignOldStatWeight (default 1.) a parameter influencing the statistical evaluation of sequence comparison. To use run-time statistics use alignOldStatWeight=0.
- Up to mnSolutions hits will be retained in the final table of hits.
- The parallel version of the program will use the number of CPUs defined by the fork command (but not more than is available in your computer). The expected time is inversely proportional to the number of CPUs.
- maxMemory is a real ICM-shell-variable defining the size of the database buffer memory in Mb used by the command. If this size is smaller than the database, the sequences will be loaded in chunks.

The output table looks like this and contains the following fields:

```
#>T SR
#> NA1 NA2            MI     MX   LMIN    LN      H      ID      SC      pP   DE
1hiv_a POL_HV1H2      57    155     99  0.665  0.099  100.0  103.69   30.00 "POL PROT..
1hiv_a POL_HV1BR      69    167     99  0.664  0.098   99.0  103.62   30.00 "POL PROT..
<i>... lines skipped ...</i>
1hiv_a POL_MLVAV       9    102     99  0.648  0.078   27.3   23.41    5.31 "PROTEASE..
1hiv_a VPRT_MPMV     172    272     99  0.799  0.290   29.3   21.95    4.82 "PROTEASE..
1hiv_a VPRT_SRV1     172    269     99  0.799  0.290   27.3   21.65    4.72 "PROTEASE..
1hiv_a GPDA_RABIT     33    145     99  0.785  0.264   28.3   20.98    4.50 "GLYCEROL3P
```

- **NA1** - the query sequence (a single command can search several query sequences)
- **NA2** - the name of the database sequence
- **MI** : MX - the matching fragment boundaries in the database sequence
- **QMI** : QMX - the matching fragment boundaries in the query sequence
- **LMIN** - the shortest sequence length in a pair (query, database sequence)
- **LN** - log-correction factor (not used in pP but you may want to use it to resort the table).
- **H** - the fraction of the database sequence covered by the alignment with the query. If you search against a database of domains this number should be close to 1 (e.g. the hit is less significant if your query is only a part of a domain). It can be taken into account by multiplying pP by this number.
- **ID** - percent sequence identity (number of identical residue pairs in the alignment divided by LMIN)
- **SC** - normalized alignment score which is used to calculated the Probability. The score depends on the residue substitution matrix and gap penalties. (see the Score function).
- **pP** = $-\log_{10}$(Probability)

- ♦ **DE** - the database sequence definition

Examples:

```
s_searchDB = s_icmhome + "/data/blast/pdbseq"
read sequence "GTPA_HUMAN.swi"
find database s_searchDB output="gtpa1"
find database pattern="C?[DN]?\{4\}[FY]?C?C" s_searchDB margin = 5

unknown1=Sequence("TTCCPSIVARSNFNVCRLPGTPEAICATYTGCIIIPGATCPGDYAN")
find database exact unknown1 s_searchDB margin=1
```

find database fast : fast dictionary-based sequence search

`find database fast` [= *i_speed*] *sequence s_dbFile* [`output=`*s_file*] [`name=`*s_tabName*]

very fast dictionary-based sequence search algorithm. Requires a blast-formated database file.
Options:

- ♦ `fast=` *i_speed* # a number from 1 (slow, rigorous) to 100 (fast and only almost identical sequences).
- ♦ `output=` *s_file* # saves the output to a file.

Example:

```
read sequence swiss web "1433B_HUMAN" # read one sequence
find database fast=90 1433B_HUMAN     # search pdbseq database (the default)
```

See also: `write index sequence` command that creates BLAST files from a FASTA file.

find molecule: chemical substructure search

A family of chemical substructure identification commands:

- ♦ `find molecule`
- ♦ `find molecule sstructure` [`tether`] *ms1 ms2* [`all`] # maximal common substructure, equivalent atom pairs in `S_out`

`find molecule` *s_Smile1* { *s_Smile2* | *S_Smiles2* } [`atom`] [`bond`] [`simple`]
Identify a complete match of the source molecule represented by a `smiles` string in another smiles string or an array of smiles strings representing a database of chemicals. Make sure that you unselect hydrogens in your smiles string.
Options:

`atom`	allow superpositions of all atom types
`bond`	allow superpositions of all bond types
`reverse`	searches

The following setup is optional:

- ♦ prepare the target strings with the `Smiles( a_//![hdt]* )` function (exclude hydrogen, deuterium and tritium)
- ♦ search the source string made without hydrogens

Only up to `mnSolutions` hits will be retained in the final table of hits. Change this shell variable if necessary. The function will return the results in the following variables:

- ♦ `i_out` - contains the number of hits
- ♦ `I_out` - contains the integer array of the hit numbers

WARNING: This is obsolete way of chemical substructure searching. Use: `find table Index chemical other chemical functions`

`find molecule reverse` *ms_1 s_smile*

WARNING: This is obsolete way of chemical substructure searching. Use `find chemical` instead.
: find molecule sstructure [all] [tether]

find **maximal common substructure** in selected atoms of the two molecules. Without the `all` option, one largest pair of matching fragments is identified. The pairs of equivalent atoms separated by a vertical bar will be stored in `S_out`, e.g.

```
a_C2H6.m/1/c1|a_C2H6O.m/1/c1
a_C2H6.m/1/c2|a_C2H6O.m/1/c2
```

Options:

- `all` : finds multiple matching fragments. Takes fragments with number of atoms >= `minMCSFragmentSize` (3 by default)
- `tether` : tethers the matching *as_molIcmObj2* atoms the equivalent atoms of *as_molObj1* . works only for the ICM_type objects (see `convert` and `convertObject`)

The **tether** option is useful since once the tethers are established, you can `superimpose a_//T` according to these tethers, or optimize the molecule with tethers, e.g.

```
build smiles "C(=CC=C(C1)CN(CCNC2)C2)C=1"
build smiles "C(=CC=C(C1)N(CCNC2)C2)C=1"
ds a_*.
find molecule sstructure all tether a_1. a_2.
superimpose a_      # uses tethers to superimpose a_ on a_1.
```

The tethers can also be interactively edited (see `delete tether` command)

`find molecule` [`tether`] *as_subFragmentQuery as_IcmTargetContainingQueryFragment*
You can also use the alternative set of arguments and use molecular selections instead of the smiles strings. The atom pairs of *as_IcmTargetContainingQueryFragment* aligned to each sequential atom of the query molecule will be stored in the `S_out` array. The atom selection of the target will also be returned in the `as_out` selection.
The `tether` option for ICM objects: After the equivalent sets of atoms in two molecules are identified, tethers can imposed pulling atoms of *as_IcmMolContainingQueryFragment* to the equivalent atoms of passive atoms *as_subFragmentQuery* . The matching atoms of the second selection *as_IcmMolContainingQueryFragment* which are pulled by tethers to the *as_subFragmentQuery* template positions can be superimposed with the `minimize "tz"` *v_positionalVariables* command.
Important: unselect the hydrogens to speed up the matching procedure, e.g.

```
  find molecule a_1.//!h* a_2.//!h*
```

An example:

```
 build string name="a" "se nter his cooh"   # query template
 build string name="b" "se nter his trp cooh" # target
 find molecule a_a./his/cg,nd1,ce1,ne2,cd2,cb,ca,n a_b. tether
 display as_out xstick # the tethered atoms of the target
 display a_*.
 minimize "tz" a_b.//?vt*
 show Rmsd( a_b.//* )  # will show the RMSD of the equivalent atoms
```

See also: `Smiles` function and the `build smiles` command.

---

find chemical: finds SMARTS pattern in 3D

`find chemical` *ms_sel s_smarts* [`all`]

Searches using smarts pattern `s_smarts` in `ms_sel.` The result (matched atoms) will be stored in `as_out` With `all` option it'll find all possible matchings.

Example:

```
 build smiles "CC1=NN=C(NS(C(C=CC(N)=C2)=C2)(=O)=O)S1" name="sulfamethizole"
 display wire
 find chemical a_ "a" all  # finds all aromatic atoms
 display xstick as_out
 find chemical a_ "[$(N~[a;r6])]" all  # finds nitrogen bonded to 6-member aromatic rin
 display cpk as_out
```

See also: `find molecule sstructure SMILES/SMARTS description`

find pdb: fragment search

`find pdb` *rs_fragment os_objectWhereToSearch s_3D_align_mask* [*s_sequencePattern* [*s_SecStructPattern*]] [*r_RMSD_tolerance*]
Find a fragment (e.g. a loop) with certain geometry, sequence and/or secondary structure.
Arguments:

- *rs_fragment:* the search fragment template
- *os_objectWhereToSearch:* the other object.
- *s_3D_align_mask:* marks the residues to be used in the 3D superposition and comparison in terms of *r_RMSD_tolerance* (see below). The number of **'x's** (or 'ON' bits) in the mask must be equal to the query fragment length (it may be discontinuous), while the total mask length should be equal to the found fragment length. For example, if you search for an 11-residue loop with the same geometry of 3-residue ends, but any geometry of the middle part your mask must be `"xxx-----xxx"`. If you want to match geometry of the middle part you would invert the mask: `"---xxxxx---"`, etc.
- *s_sequencePattern:* Use `"*"` for any sequence. Otherwise you may use regular expressions, for example: `"?A[!P]???$"`.
- *s_SecStructPattern:* Use `"*"` for any secondary structure pattern. Otherwise, specify a regular expression, for example "?HHH___EE[!_]".
- *r_RMSD_tolerance:* RMSD threshold to accept a fragment as a solution. To avoid time-consuming `optimal 3D superposition` during the search, **distance Rmsd** (i.e. root-mean-square deviation between two Ca-atom distance matrices of the compared fragments) is used as a measure of spatial similarity on the preliminary stage of each comparison. However, in the resulting list of hits, collected in *SearchSummary* `string array` the optimal 3D superposition coordinate `Rmsd` is presented. Therefore, RMSDs in the output list may exceed the specified threshold.

Hits will be stored in `s_out` . The following just illustrate the syntax, it does not make much sense, since you need to loop through a database of objects to find something interesting.
Example:

```
read pdb "1crn"
read object s_icmhome +  "complex"        # object in which to search
find pdb a_1./16:18,20:22 a_2. "xxx----xxx" "V[LIVM]?????G??" "*" 2.5
print s_out
```

find prosite or profile

`find prosite` [`append`] *seq* [*r_minScore*] [*i_mnHits*]
find matching `prosite` patterns, store results in the SITES `table` . Option `append` indicates that the results should be appended to the existing SITES table. The default *r_minScore* is 0.7 . The default *i_mnHits* is defined by the `mnSolutions` parameter.
Examples:

```
read sequence "zincFing.seq"
find prosite lznf_m
show SITES
```

find pattern

`find pattern` [`number`] [`mute`] *s_sequencePattern* [*i_mnHits*] [{ *os_objectWhereToSearch* | *seq_Name* | *s_seqNamePattern* } ...]
find specified sequence pattern (i.e. "[AG]????GK[ST]" for ATP/GTP-binding site motif A) in ICM shell sequences or molecular objects. Hits will be stored in `s_out` . `r_out` contains the number of found hits divided by the expected number of hits, as suggested by random distribution of amino-acids with frequencies from the Swissprot database. This "found/expected ratio" is also reported if `l_info=yes`. If this number is 1. it does not mean anything, 10. means that you can publish the finding and two paragraphs of speculations, 10000. means that somebody else has already found this hit. Pattern language:

- ^ sequence beginning
- $ sequence ending
- ? one character
- * any number of any characters
- [ACD] alternatives
- [!ACD] all but the specified residues
- *char* \{ *i_min*, *i_max* \} : repetition. E.g. ?\{5,8\} from 5 to 8 of any character.

Other arguments and options:

- ♦ `number` - just report the number of hits instead of reporting each match
- ♦ `mute` - suppress terminal output (used in scripts)
- ♦ *i_mnHits* - (default `mnSolutions`)
- ♦ *os_objectWhereToSearch* - the target molecular object.
- ♦ *seq_Name* - the target sequence. By default, the search is performed among **all** currently loaded sequences.
- ♦ *seqNamePattern* - the target sequence name pattern to search through many sequences loaded to the shell.

Returned values

- ♦ `s_out` - text output of all matches
- ♦ `i_out` - the number of hits
- ♦ `r_out` - the ratio to the random expectation (it r_out>1. it means that the number of hits is larger than the random expectation).

See also the `searchPatternPdb` macro.
Examples:

```
read sequence s_pdbDir + "/derived_data/pdb_seqres.txt"  # all pdb-sequences
find pattern "[AG]????GK[ST]"   # search for ATP/GTP-binding sites
searchPatternPdb "^[LIVAFM]?\{115,128\}[!P]A$"
    # ^ : seq.start; ?\{115,128\} from 115 to 128 of any res.; $ : seq.end
```

See also: `read prosite`, `s_prositeDat` .

---

find molcart : chemical search in Molcart database

`find molcart` [`sstructure`|`similarity`|`exact`] `table=`*s_molcartTable* [*s_smarts*|*S_smarts*|*chemarray*] [`r_distCutOff`] [`only`] [`stereo`] [`name=`*s_resultName*] [`query=`*s_SQL_condition*] [`output=`*s_molcartTable*] [`number=i_maxHits`] [`exclude=s_smarts`|`S_smarts`] [`append`|`delete`] [*connection_options*]

Performes chemical search in Molcart database. Connection may be specified by *`connection_options`*

Supported search modes are:

- ♦ `exact` : exact match
- ♦ `sstructure` : substructure
- ♦ `similarity` *r_dist* : find similar compounds with distance cutoff *r_threshold* (between 0 and 1, e.g. 0.1 for very similar compounds)

Other options:

- ♦ `append` : the search results will be appended to the output table, if it exists.
- ♦ `delete` : the specified output table will be overwritten if it already exists.
- ♦ `center` : performs a K-means clustering of all chemicals in the specified table and selects a representative.
  - ◊ `number=`, or
  - ◊ `distance=`
- ♦ `query` = *s_SQL_condition*> using existing table columns or on-the-fly built-in functions (e.g. MolPSA) in an sql expression., e.g.

```
find molcart table="amri" query="MolLogP(t.mol)<2 and MolAtomCount(t.mol)=10"
   # t is a generic name for all tables
```

  allows one to use the following built-in functions in sql-style expressions (see above) : MolAtomCount, MolFormula, MolHBA, MolHBD, MolLogP, MolLogS, MolMaxFusedRings, MolMaxRingSize, MolMinRingSize, MolNofMol, MolNofRings, MolPSA, MolRotB, MolSmiles, MolVolume, MolWeight, MoldHf . It also allows explicit fields of the specified table to be mentioned as well, e.g. `query="t.molid=2345"` In SQL allowed logical and comparison operators are `and`, `or not` , =, > < ≥ ≤ !=

- ♦ `output` : The `output` option allows one to save search results in another database table *s_molcartTable*. It is possible to specify a table in another `molcart` connection by using "connectionID;database.table" format.

Examples :

```
  find molcart sstructure table="pub.all" "c1ccccc1" number=1000 name="myHits"  # by su
# find by substructure (contains 4 benzene rings)
  find molcart sstructure table="pub.all" "c1ccccc1" query="MolNofRings(t.mol)=4" name=
  find molcart exact table="pub.all" t.mol  # finds exact matches. chemical array patte
  find molcart similarity 0.2 table="pub.all" "CC1=CN(C(NC1=O)=O)[C@H]1C[C@@H]([C@H](CC
```

See also: `Index chemical` `Nof` `Find` `find table`

find table : chemical search in ICM table

`find` `table` {*T_table*|*filename=<s_file*} {`sstructure` [`group`]|`similarity`|`exact`} [*s_smarts*|*S_smarts*|*chemarray*] [`r_distCutOff`] [`only`] [`stereo`] [`query=`*s_ICM_condition*] [`name=`*s_resultName*|`select`] [`index=I_index`]] [`append`]

Performs chemical and text search in the local table.

Arguments:

- ♦ *T_table* : input table.
- ♦ Alternatively an `SDF` or CSV *s_file* may be specified.
- ♦ *s_smarts* or *S_smarts chemarray* : input pattern
- ♦ search type: `sstructure` `similarity` `exact`
- ♦ *r_distCutOff* distance cutoff for similarity search
- ♦ With `stereo` option chirality will be taken into account in substructure and similarity searches.
- ♦ *group* option toggles the special search mode when all atoms in the pattern except attachment points are treated "as drawn" (not other attachments are allowed)
- ♦ With `only` option only number of hits will be returned.
- ♦ With `select` option matched rows in the original table will be selected (not result table will be created)
- ♦ `name=s_resultName` result table name (ignored with `select` option)
- ♦ `query=s_ICM_condition` extra condition. Using this argument you can specify an extra logical condition for the query. Column names, string constants and numnbers can be used: For example : MolWeightWith `append` option, search results are appended to the result table

Examples :

```
  group table t Chemical({ "CC(=O)Oc1ccccc1C(O)=O", "CC(Nc1ccc(cc1)O)=O" } ) "mol"
  add column t Mass(t.mol) name="MW"
  find table t query = "MW<160" select  # select rows with molecular weight < 160

  cc = Chemical({"CC(=O)Oc1ccccc1C(O)=O","CCCc1c2c(C(NC(c3cc(ccc3OCC)S(N)(=O)=O)=N2)=O)
  group table t2 cc "mol"
  # compare chemical tables
  find table exact t  t2.mol select   # select rows in t
  find table exact t2 t.mol  select   # select rows in t2

  find table similarity t  t2.mol select 0.5
  find table similarity t2 t.mol  select 0.5

  # Not enough? Let's increase distance cutoff

  find table similarity t  t2.mol select 0.8
  find table similarity t2 t.mol  select 0.8

  # this command can also be used to select arbitrary rows in a table
  find table t2 select index = {1 3 5}
  Index( t2 selection )
```

See also: `Index chemical` `Find` `find molcart` `SMILES/SMARTS` `other chemical functions`

find pharmacophore : pharmacophore search in ICM table

`find pharmacophore` *as_pharmQuery chemarray3D* [`all`]

Performs a pharmacophore search in *chemarray3D* using *as_pharmQuery.*

Example:

```
read binary s_icmhome + "example_ph4.icb"
find pharmacophore  a_pharma. t_3D.mol
```

all option allows one to score all possible mapping for each conformation

See also: `Rmsd` `superimpose` `makePharma`

fix

`fix` *vs*
fix (exclude from the free variable list) specified variables (such as bond lengths, angles and phases or torsions) in an ICM-object. This operation can be applied to the `current object` only (use `set object` *os_object*first). See also: `unfix` .
Examples:

```
 set v_//omg 180.               # set all omega torsions to the ideal value
 fix v_//omg                    # fix all omega torsions

 fix v_/8:16,32:40/phi,PSI,omg*  # fix the backbone in two fragments
```

Note using `PSI` torsion reference for correct residue attribution.

for

`for`
is one of the `ICM flow control` statements, used to start a `loop` in the ICM-shell. See also `while`, `endfor` .

fork

a powerful tool for parallelization of ICM-shell scripts.
`fork` [*i_nExtraProcesses*] [`pipe`]
spawns one or the specified number of extra copies of ICM. This command will only work in a non interactive mode, i.e. you should run icm like this:

```
 icm _multiProc  # from the unix shell or
 unix icm _multiProc  # from the interactive ICM-shell
```

The `Index` (`fork`) will contain the current process number, and `Index` (`fork system`) returns process id. The parent process has both values at zero.
the *pipe* option will redirect the output to the parent process and synchronously print it in the `wait` command

The simplest parallel script. Note that l_out==yes (or Index(fork)==0) defines if the script runs in the parent process.

```
#!icm64 -s
fork 4
print l_out, Index(fork), Index(fork,all), Index(fork,system)
wait
print " back to parent"
quit
#
```

An example script _multiProc with a hypothetical macro bigDatabaseJob which takes two arguments: the number of database chunks, the current chunk number, and the output file name:

```
 read libraries
 macro bigDatabaseJob i_nChunks i_Chunk s_outFile  # definition
  ...
 endmacro

 read sequence "hot"
 fork 4
            # spawn 4 extra processes, total 5
 ip = Index( fork )
 bigDatabaseJob 5  ip  "out"+ip
            # work on section ip,
            # save results to files out1 out2 ..
 wait       # also quits all extra processes
 unix cat out1 out2 out3 out4 out5 >! out.tab
 read table "out.tab"
 ....
 quit
```

Parallel processing with aggregation through the internal pipe and without file output

See also:

- ♦ `Index`(fork [system,all]) - current process index, pid, and current number of children
- ♦ `Nof`(fork) - the number of available cores in the current computer
- ♦ `wait`.

fprintf

`fprintf` [`append`] *s_file s_formatString arg1 arg1 arg2 arg3 ...*
formatted print to a file. The specifications for *s_formatString* are described in the `printf` command section. In contrast to the `print` and `printf` commands, the result of the fprintf command is not shown. E.g.

```
 fprintf        "a.txt" "%s\n"      "Day Temparature"
 fprintf append "a.txt" "%s %.2f\n" "Monday", 22.4
 fprintf append "a.txt" "%s %.2f\n" "Tuesday", 27.334
```

function

- a group of ICM commands with a name and arguments returning a shell data object. **Definition:** `function` *name* (*arg1 arg2*) *code code var* = .. `return` *var* `endfunction`

Examples:

```
function Fibo( i )
  a=0; b=1; I={1}
  while b<i
    I = I //b
    x=a; a=b; b=x+b
  endwhile
  return I
endfunction
ii = Fibo(1000)
show ii
```

Example where the function returns a collection

```
function ArrayStats( R )
  c = Collection()
  if(Nof(R)<2) return c
  c["mean"] =Mean(R)
  c["sigma"]=Rmsd(R)
  c["A"]    = 1./(c["S"]+1.e-18)  # protection against div by zero
  c["B"]    = -c["M"]
  n=Nof(R); sort R
  c["median"] = (Mod(n,2)==0)?((R[n/2]+R[(n+2)/2])/2.):R[(n+1)/2]
  return c
endfunction
ArrayStats({1. 2. 3. 4.})["median"]
c = ArrayStats({1. 2. 3. 4.})
```

see also `macro`.

global command

`global` any_ICM_command
guarantees that the new ICM shell variables created or read to the shell are at the main shell level, rather than nested inside macros.
By adding `global` to any `read` command in a `macro` you make the `keep` *variableName_or_type* command at the end of the macro unnecessary. Example:

```
macro read_alignment s_file
  global read alignment s_file
endmacro
```

goto

`goto`
is one of the `ICM flow control` statements, used to `jump` over a block of ICM-shell statements. See also `break` , `continue` .

group

group sequence

`group sequence` [`fast`] [*seq1 seq2 ...* | *s_seqNamePattern* | `alignment` | `selection`] `GroupName` [pdb] [`unique` { *i_MinNofMutations* | *r_MinDistance* } [`delete`]]
group sequences into a `sequence group` to perform a multiple alignment with the `align` command.
Option `unique` allows you to select only the different sequences. If no argument follows the word `unique`, only identical sequences will be dismissed, otherwise they will be compared and retained if the number of differences is greater than *i_MinNofMutations* (integer argument) or the `distance` between two sequences is greater than *r_MinDistance* (real argument).

Option `pdb` activates preferences for higher resolution and first chain names ('a' is better than 'b', etc.)

Option `fast` will activate the dictionary approach and will give a big time benefit for very large collections (tens of thousand or more).

The comparison criterion is complex and has the following set of preferences which may be useful in extracting a `representative subset of sequences` from a PDB-database:

- **longer** sequence is better that shorter
- with the `pdb` option, or if all the names contain the X-ray **resolution** 2 digit suffixes (like a19 and 9lyz24, for resolutions 1.9 and 2.4 respectively), higher resolution is better (1.9 is better than 2.4). **Note** Resolution suffixes are added by the `read pdb sequence resolution` command
- higher number in a pdb-file name is preferable, i.e. 9lyz is better than 3lyz. (I would not die for this principle, though).
- with the `pdb` option, identical chains have alphabetical preferences (e.g. 9lyz_a is better than 9lyz_b).

Suboption `delete` tells the program to delete from ICM-shell all the sequences which were found redundant by the `unique` option.
Examples:

```
read sequences s_icmhome+"seqs.seq" # load sequences
group sequence aaa                  # group ALL the sequences into aaa
group sequence seq3 seq1 seq2 aaa   # explicit version of the previous line
align aaa                           # multiple alignment

read sequences s_icmhome+"azurins.msf" # some of sequences are very close
                                    # but not identical
group sequence myAzur unique fast 0.15   # 0.15 is a Dayhoff-corrected minimum
                                    # intersequence distance threshold
group sequence myAzur unique 26     # all sequence pairs differ in
                                    # more than 26 positions
group sequence myAzur unique delete # duplicates will be removed
```

group sequence unique: clustering, redundancy removal and assembly

`group sequence unique= "nt,junk,simple,overlap[ > nRes ]"` [*i_wordLen*=6 [*i_dictDepth*=10 [*i_nofMutations*=0]]] [`delete`] [`nosort`] [*seq1 seq2 ...* | *s_seqNamePattern* | `alignment`] `GroupName`

If you read a very large **redundant** set of sequences and sequence fragments some of which may (i) overlap or (ii) be included in another sequence, you may want to remove all the redundant fragments, and merge the overlapping sequences into a smaller number of longer sequences. In a simple case, if the number of sequences is not too large (less than a few hundred), this removal of redundancies and fragments in your sequence set, can be performed with the `group sequence unique ..` command described in the previous section.

To work on much larger sequences sets and allows one to merge overlapping sequences a more advanced algorithm is needed. This ultra-fast removal of redundant protein or DNA sequences, may also assemble the sequences into larger consensus sequences and is invoked by `group sequence unique=` *"options"* command.

The command returns the result as a `sequence group` *GroupName*. Will work on tens of thousands of sequences at once. The important features of the command:

- ♦ it can cluster/unique millions of sequences very quickly (your computer just needs enough memory).
- ♦ larger sequences incorporate the matching smaller ones.
- ♦ merged or absorbed sequence names are added to the description of their master unless `nosort` is specified
- ♦ merging (option "overlap") protein sequences requires sticky C-terminus letter 'X'
- ♦ the algorithm is based on a dictionary approach and allows one to have mismatches
- ♦ matching rules:
  - ◊ for proteins: any letter matches 'X', B=(D or N) and Z=(E or Q)
  - ◊ for nucleic acids: any letter matches 'N'
  - ◊ if possible, 'X','N','B','Z' are replaced by a more specific letter from the matched sequence

Options (they can be combined in a comma-separated string, e.g. "`nt`,simple,junk"):

- ♦ `delete` - the non-unique sequences are deleted not only from the group but also from the shell
- ♦ `nosort` - do not merge descriptions of the merged sequences
- ♦ `unique= "simple"` - the fastest mode. It will eliminate `only` the exact duplicates.
- ♦ `unique= "nt"` means that DNA or RNA sequences are compared (the program assumes protein sequences by default and a corresponding *i_wordLen* of six). This implies the alphabet of A,C,G,T (or U) and the word length should therefore be increased. The default nucleic acid sequence word length of 13 allows one to fit the entire dictionary into the memory of 256 Mbyte. If your computer has less memory, reduce the *i_wordLen* to a smaller value.
- ♦ `unique="junk"` this option tells the program to remove sequences that do not contain any meaningful sequence. This means that they are mostly composed of 'X's or 'N's and the intermittent sequence is shorter than *i_wordLen.* This option is almost always useful.
- ♦ `unique="stripX"` this option tells the program to strip X (or N for nucleotide) character stretches from the beginning and from the end of the sequence. Those will be compressed into just one character. Useful if your sequences were *dusted* or repeat-masked.
- ♦ `unique="noX"` this option tells the program to skip the sequence quality enhancements (replacement of 'X','N','B','Z' by a more specific letter from the other very similar sequence).
- ♦ `unique="complement"` with this option the complementary nucleic acid sequences will also be considered and removed if redundant. This option **can not be combined with the "overlap" option**.
- ♦ `unique="overlap[>numberOfRes]"` Merge overlapping fragments in in addition of deleting the subfragments from the set. The number of overlaping nucleotides or amino acids can be redefined, e.g. `unique ="overlap>25"`
  - ◊ Two **amino acid sequences** are merged only if there is the overlap is greater than the threshold (12 amino acids by default) and the overlapping C-terminal residue is 'X'. An example of the allowed merge for protein sequences:

```
s1     VTIKIGGQLKEALLDXGADDTVLEEMSLPGX-------
s2     ----------EALLDTGADDTVLZEMSLPGRWKPKMIG
result VTIKIGGQLKEALLDTGADDTVLEEMSLPGRWKPKMIG
```

If for some reason your ESTs do not terminate with 'X's, they can be added by the following procedure:

```
for i=1,Nof(sequence )
  sequence[i] = sequence[i] //Sequence("X")
endfor
```

◊ Two **nucleic acid sequences** are merged if the overlap is 30 by default. There are NO special requirements for an 'X' nucleotide flanking the sequences.

- ♦ *i_wordLen* (6 by default, 14 if the `unique="nt"` option is specified). The length of a word in the dictionary. The memory occupied by the dictionary depends exponentially on his length.
- ♦ *i_dictDepth* (10 by default) limits the number of sequence fragments referenced referenced from a single 'word'. This option prevents the dictionary from growing to much in memory (what the product of *i_wordLen* * i_dictDepth) .
- ♦ *i_nofMutations* (zero by default) the maximal number of mutations/mismatches between sequences which are considered to be redundant.

See also:

- ♦ `Trans(` *seq_* frame) - to translate a DNA sequence
- ♦ `align new` # to align a cluster and generate the consensus
- ♦ `Find(` *sequence* , *s_keyword*) # to find a retired sequence with the *s_keyword* in its title among the newly formed sequences
- ♦ `show [color]` *ali_*

---

group table

`group table` [`copy`] [*u_name*] [*array* [*s_name*] ..] `header` [*sh_obj s_name* ..]

WARNING: The `append` option of this function is obsolete. Use `add column` instead.

create a new `table` from individual arrays or append new columns or table header elements to an existing table. This example shows how an ICM table including both header elements and columns may look like:

```
group table t {1 2} "a" {"one","two"} "b" header "trash" "comment" 2001 "year"
 Info> table  t  (2 headers, 2 arrays) has been created
  Headers: t.comment t.year
  Arrays : t.a t.b

show t
 #>s t.comment                     # TWO HEADER ELEMENTS
 trash
 #>i t.year
  2001
 #>T t
 #>-a-----------b----------        # TABLE ITSELF
    1           one
    2           two
show t.comment t.year
 trash
   2001
```

Options:

- ♦ `copy`: make a copy of the original ICM-shell object and move it to the table.
- ♦ `append`: add specified ICM-shell objects to the table (default: overwrite) # not recommended.

In the header section each ICM-shell object should be followed by a string specifying the variable name. The empty string will be interpreted as an indication to keep the name of the variable. Unnamed constants such as {1 2 3} or "adsfasdf" will be automatically assigned unique names.

See also: `split`, `Table` , `add column` (to append columns) .
More examples:

```
 a=1   # integer a
 b=2.  # real b
 group table copy t header a "ii" b "rr"
       # create table t with t.ii and t.rr header objects
 show t
 group table t header a "" b ""
```

```
      # t with t.a and t.b header objects
show t
group table t {1 2 3} {2. 3. 4.}
      # t with automatically named table
      # arrays t.1 and t.2
show t
group table t {1 2 3} "a" {2. 3. 4.} "b"
      # t with table arrays t.a and t.b
show t
split t # split the table into individual arrays
```

See also: `group by column` , `split` aggregated cells in a column by a separator.

group table by column with non-unique values

`group` *t.keyColumnToGroupBy* [{*t.extraColumn*|--`all`} [`s_colRule`[,`colname`]]] ... [`separator=`*s_sepString*]
sorts and groups a table by unique values of the key column *t.keyColumnToGroupBy* . Then applies the specified extra column value combination rules (or functions).
The following column cell merging rules can be applied to the numerical arrays:
`"uniq"|"mean"|"min"|"max"|"first"|"last"|"rmsd"|"sum"`
The string arrays can be grouped with the following subset of the above functions:
`"uniq"|"min"|"max"|"first"|"last"`
The `"uniq"` function is the default, and it means that the unique column values with the same *key* field will be accumulated by the group command.

Function	Description	Array Type
`uniq`	merge unique field values into v1,v2,v3	I,R,S
`first`	keep the *first* value in the grouped table	I,R,S
`last`	keep the *last* value in the grouped table	I,R,S
`mean`	find the *mean* value with the same key	I,R
`min`	find the *minimal* value with the same key	I,R,S
`max`	find the *maximal* value with the same key	I,R,S
`rmsd`	find the *root-mean-square* deviation of values with the same key	I,R
`sum`	find the *sum* of values with the same key	I,R

Example:

```
group table t {1 2 1 1 2} {1. 2. 3. 4. 5.}
t
 #>T t
 #>-A-----------B----------
    1          1.
    2          2.
    1          3.
    1          4.
    2          5.

group t.A    # groups in place
t
 #>T t
 #>-A-----------B----------
    1          1.,3.,4.
    2          2.,5.
split t.B separator=","   # opposite operation in place, converts to rarray automatical

group table t {1 2 1 1 2} {1. 2. 3. 4. 5.}
group t.A t.B "sum,C"  # sum t.B values and call the column C
#>T t
#>-A-----------C----------
   1          8.
   2          7.
```

There are two special rules `"refmin"` and `"refmax"` which can be applied in conjunction with `"min"` and `"max"` and take rows corresponding to minimum or maximum values in the group.

Note that specifying `all` option instead of column name will apply operation for all the rest of columns.

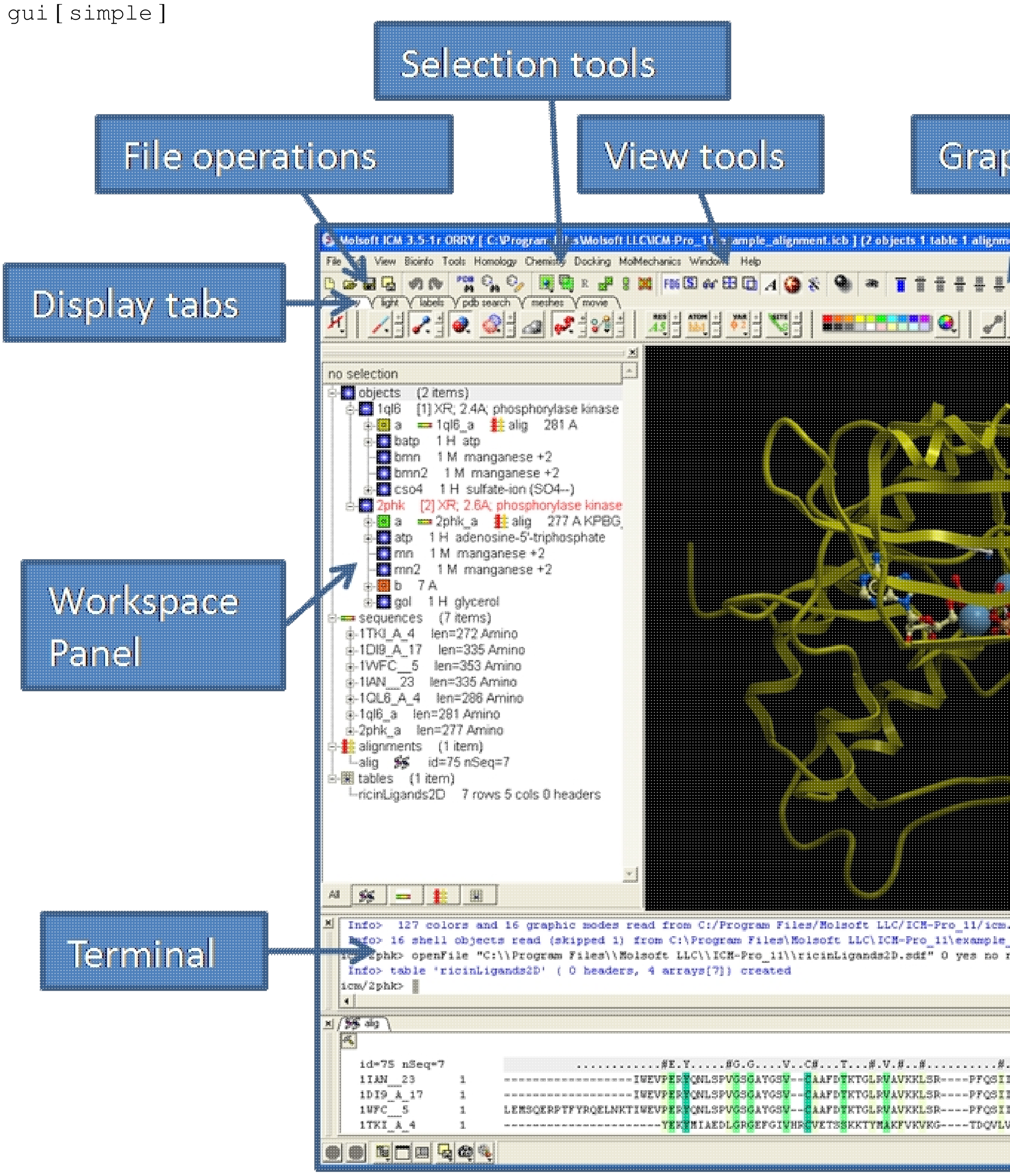
Examples:

```
group table t {1 1 2 2} {2 1 3 4} {"a" "b" "c" "d"} {"a" "b" "c" "d"}
group t.A t.B "min,B"  all "refmin,C"  name="t1"
group t.A t.B "max,B"  all "refmax,C"  name="t2"
```

See also: `split` aggregated cells in a column by a separator.

GUI and Programming Dialogs in ICM

`gui [ simple ]`


Selection tools
File operations
View tools
Grap
Display tabs
Workspace Panel
Terminal
File Edit View Bioinfo Tools Homology Chemistry Docking MolMechanics Windows Help
light labels pdb search meshes move

```
no selection
objects   (2 items)
 1ql6   [1] XR; 2.4A; phosphorylase kinase
  a   1ql6_a   alig   281 A
  batp   1 H  atp
  bmn   1 M  manganese +2
  bmn2   1 M  manganese +2
  cso4   1 H  sulfate-ion (SO4--)
 2phk   [2] XR; 2.6A; phosphorylase kinase
  a   2phk_a   alig   277 A KPBG
  atp   1 H  adenosine-5'-triphosphate
  mn   1 M  manganese +2
  mn2   1 M  manganese +2
  b   7 A
  gol   1 H  glycerol
sequences   (7 items)
 1TKI_A_4   len=272 Amino
 1DI9_A_17   len=335 Amino
 1WFC__5   len=353 Amino
 1IAN__23   len=335 Amino
 1QL6_A_4   len=286 Amino
 1ql6_a   len=281 Amino
 2phk_a   len=277 Amino
alignments   (1 item)
 alig   id=75 nSeq=7
tables   (1 item)
 ricinLigands2D   7 rows 5 cols 0 headers
```

```
 Info>  127 colors and 16 graphic modes read from C:/Program Files/Molsoft LLC/ICM-Pro_11/icm.
 Info> 16 shell objects read (skipped 1) from C:\Program Files\Molsoft LLC\ICM-Pro_11\example_
icm/2phk> openFile "C:\\Program Files\\Molsoft LLC\\ICM-Pro_11\\ricinLigands2D.sdf" 0 yes no
 Info> table 'ricinLigands2D' ( 0 headers, 4 arrays[7]) created
icm/2phk>
```

```
  id=75 nSeq=7              ..............#E.Y.....#G.G....V..C#...T...#.V.#..#..........#.
  1IAN__23       1    -------------------IWEVPERYQNLSPVGSGAYGSV--CAAFDTKTGLRVAVKKLSR----PFQSII
  1DI9_A_17      1    -------------------IWEVPERYQNLSPVGSGAYGSV--CAAFDTKTGLRVAVKKLSR----PFQSII
  1WFC__5        1    LEMSQERPTFYRQELNKTIWEVPERYQNLSPVGSGAYGSV--CAAFDTKTGLRVAVKKLSR----PFQSII
  1TKI_A_4       1    ----------------------YEKYMIAEDLGRGEFGIVHRCVETSSKKTYMAKFVKVKG----TDQVLV
```

start menu-driven graphical user interface from command line. The GUI runs the icm.gui file containing all the commands invoked by menus or pop-ups.
Option `simple` allows you to keep your terminal window separate from the graphics window.

```
% icmgl
 gui simple
```

You can also invoke gui from the command line, e.g.

```
icm -g # or
icm -g mymenus.gui
icm -G mymenus.gui  # keep the original terminal window
```

Terminal window and fonts
`icm -g` (or `gui` command) invokes a GUI frame *with its own built-in terminal window* . To influence the font size in this terminal window, modify `XTermFont` record of the `icm.cfg` configuration file, e.g.

```
XTermFont *-fixed-medium-*-*-*-24-*
```

If you prefer to keep the original terminal window use the `icm -G` option or invoke the `gui simple` command from ICM shell (I keep `alias guis gui simple` in my personal configuration file). In this case you can change the font of the terminal window with standard means of the window manager.
3D Graphics window
GUI has a GL-graphics window which can be undisplayed with the

```
 undisplay window
```

command or from GUI by choosing `Clear/No_graphics` menu item.

See also: `gui programming`

help

get help from `icm.htm` file. Set `s_helpEngine` variable to "icm" (internal help in the text window), "netscape" or any other web-browser. **Important**, make sure that the `s_webViewer` variable points to your html-browser (e.g. `s_webViewer = "firefox"`), see also **File/Preferences/DisplayGeneral** .
Getting help in built-in ICM html browser.

`help` *command|function|icmVariable|s_icmHelpAncorName*

`help` "`I:`*anchorName*"

open the specified section in the ICM Language Reference Manual

`help` "`G:`*anchorName*"

open the specified section in the ICM Program Guide

`help` *s_htmlFileName* # the file name must be followed by the pound sign

opens a single html file in the built-in ICM html browser. This file may contain sections of icm script in the following format:

```
<!--icmscript name="action1"
read pdb "1crn"
display a_*.
-->
...
<a name="action1" href="#action1">click here to execute icm script</a>

help read pdb  # opens the read-pdb section
help "G:learning"
help "I:montecarlo"
help "myfile.html#"
```

`help` *command|function|icmVariable|icmHelpAncorName*

help

 `help` [`input=` *s_fileName*] [*word1 word2* ...]
get full help in either text or html form, redirect it to the specified file, if the `input` option is specified. Do **not** use plural forms of the nouns. Examples:

```
 help Random
 help read sequence
```

The built-in help engine does not know about keywords. It is recommended to use the on-line version of the ICM manual which has a well-developed Index (download the newest version of the manual, `man.tar.gz` from the Molsoft ftp site).

help commands

`help commands` [*s_Pattern*]
generates concise list of syntax lines for all or specified commands.

help functions

`help functions` [*s_Pattern*]
generates concise list of syntax lines for all or specified functions.
Examples:

```
 help                      # type  /stereo, and then letter n or Bar

 help help                 # how to get help

 help commands             # list syntax of all commands

 help commands  "rea*"     # list syntax of all read commands

 help functions            # list syntax of all functions

 help functions Matrix     # list syntax of the Matrix family of functions

 help                      # start the browser to use its own search means
 help montecarlo           # just the command name
 help real constant

 help read pdb

 help Split
```

history

`history` [`unique` | `full`] [*i_NumberOfLines*]
display previous commands. Option `unique` squeezes out the repetitive commands. Without the `full` option the commands executed from the file (rather than manually typed) will not be shown. The `unique` option hides the repetitions of the same command.

For example:

```
 history 20              # show last 20 lines
 history unique
```

To delete all previous history lines, use the `delete session` command. In this case the `write session` command will save only the new history lines.

if

`if`
is one of the `ICM flow control` statements, used to perform `conditional statements`. See also: `then`, `elseif`, and `endif` .

info

`info auto write`

Shows information about when autosaving was performed in the current session.

Database additional statistics

`info` `molcart` [*connection_options*]

Prints additional information about the Molcart connection. Connection may be specified by *`connection_options`*

See also: `molcart`

keep

`keep` `ICM-shell-variable-name1` .. [`global`]
retain specified ICM_shell variables or their classes (e.g. real, rarray etc.). This command is used in macros to avoid automatic deletion of all the local ICM-shell variables.
Also note that four classes of standard ICM-shell variables, `reals`, `integers`, `logicals`, and `preferences`, are automatically restored to their initial values by default. You can use the keep command to retain their new values.

Examples:

```
 macro rdseq s_pdbName  # extract sequence from a pdb-file
   read pdb sequence s_pdbName
   rz = Resolution(s_pdbName,pdb)
   mncalls = 10         # the existing standard shell variable
   keep rz, sequence    # retain all the sequences and rz
   keep mncalls         # retain its new value
 endmacro
```

Note that by default values will be kept only for the one level higher. With *global* option changes are propogated through the all nested levels to the global namespace.

Example :

```
s_a = "global"
s_b = "global"
macro m1
  m2
  print "m2: ", "s_a =", s_a, "s_b=" s_b
endmacro

macro m2
  s_a = "m2"
  s_b = "m2"
  keep s_a   # will be kept only for m1
  keep s_b global  # will be kept globally
endmacro

m1
print "global: ", "s_a =", s_a, "s_b=" s_b
```

join tables

`join` [`left`|`right`] *T1.col T2.col2* [`name=` *s_newTableName*] [`column=`*S_outputColumns*] [`stereo` `off`]

Unites some or all data of the two tables into another table. If the *s_newTableName* coincides with the one of the tables, the new table will replace it. The default output table name is `T_join` .

The main two arguments are two columns *T1.col1* and *T2.col2* with matching values. You can use chemical structure column to join by exact structure match. `stereo off` option can be added to ignore chirality.

The `column=` argument contains the list of column names to be retained in the output table.

Columns in the new output tables. The columns for the output table can be listed as the `column=` By default action is to include all columns from both tables. The columns by which the tables are joined will turn into one, therefore the total number of columns by default will be `N1+N2-1`.

Column names of in the joined table. The column takes are preserved unless they collide (i.e. `T1.B` and `T2.B` are both present). The the latter case the first column retains its name while the column from the second table will be named *T2name.colName* , (e.g. `T_join.B` , `T_join.T2_B`).

Types of the join command There are three types of the join command:

- inner join - the default mode, no keyword needs to be specified. The **inner join** returns all rows from both tables where there is a match in the order of the *T1.col1* column. If there are rows in *T1.col1* that do not have matches in *T2.col2*, those rows will not be included in the output column. This table can easily be empty, if the values do not overlap. The number of rows of the output table is less or equal to the number of rows in the first table. Example:

```
group table t1 {1 2 3} "A"  # 1 has no match in t2
group table t2 {"a" "b" "c"} "A" {2 3 4} "B"
show t1,t2
 #>T t1
 #>-A----------
    1
    2
    3
 #>T t2
 #>-A-----------B----------
    a           2
    b           3
    c           4

join t1.A t2.B name="t3"
t3
 #>-A-----------t2_A-------
    2           a
    3           b
```

- `left`The **left join** returns **all** the rows from the first table, extended with the matching rows from the second table. For *T1.col* rows there with no matches in the second table, empty fields will be added. The number and order of rows of the output table is equal to the number of rows in the first table. Example:

```
 group table t1 {1 2 3} "A"  # 1 has no match in t2
 group table t2 {"a" "b" "c"} "A" {2 3 4} "B"
 join left t1.A t2.B
 T_join
 #>T T_join
 #>-A-----------t2_A-------
    1           ""
    2           a
    3           b
```

- `right`the **right join** returns all the rows from the second table, and appends fiels from the first table if a match is found. It is identical to the `left` join but with two arguments swapped (`join left t1.A t2.B` is the same as `join right t2.B t1.A`). Example:

```
 group table t1 {1 2 3} "A"  # 1 has no match in t2
 group table t2 {"a" "b" "c"} "A" {2 3 4} "B"
 join t1.A t2.B right name="ttt"
 ttt
 #>T ttt
 #>-A-----------B----------
    a           2
    b           3
    c           4
```

- `local`the **local join** returns all the rows from the first table. Values in the matching (by name) columns will be overwritten with values from the second table for matched rows. Example:

```
add column t1 {1 2 3} {1 2 3}
add column t2 { 2 3} {4 5}
join t1.A t2.A local name="t1"
```

See also the `add table` (command for appending a table with identical column structure) `add column` or `add column function` (adding new columns)

learn from a training data set and create a predictive model

`learn` *t.Y* | { *Y t* } | {*Y t M*} `[all]` [*options*]

learns how to predict column *t.Y* from other columns or a matrix using the specified method; creates a modelobject.

Options:

`type="pls"|"pcr"|"kernel"|"nn"`

- the training method: partial least squares, principal component regression, kernel regression, or nearest neighbor

`kernel="dot"|"polynomial [iOrder C0]"|"radial [exp]"|"tanimoto"|"sigmoid [K C]"`

`name=` *s_outputModelName*

`column=` *S_columnNames*

- an array of column names

`all`

- forces to use all numerical columns in addition to the chemical column

`test` [= *nCross*|*I_excludedTestRows*] `# cross-validation group number or test rows`

`center # enforce the constant @@{w,,0} (see below) to be **zero .`

`select=` *R_2_c_eps*

`select=` *I_LatentVectorSetForPls*

this command takes a real array *Y* and a matrix or table of descriptors and builds an optimal cross-validated predictive **model** for property *Y*. This command can build several different types of models:

- Partial Least Square model (PLS-regression) in which $Y_i = w_0 + Sum(wi*X_{ij})$
- Principal Component Regression (PCR) which is a similar linear model as PLSR, but identified and build in a different way.

The **output** of this command is the following:

- a predictive model object (one-element `parray` of subtype 'model'. See also `Parray(` model *s_name*))
- a new *Ypred* column with self-predicted values is added to the training set table
- a new *Yprex* column with cross-validated values is added to the input table
- rmsError and correlation coefficient for Self- and Cross-validated (CV) predicted values (see the example) in R_out.

Example:

```
read table "t.csv"
learn t.A
 learn t.A
 Info> plsRegression model for property 'Apred' built for 95 records.
 Corr_R2=0.44 (CV=0.36),  rmsError=0.48 (CV=0.51)

learn t.pK test=4 select={2,10,20,30} method="pls"
```

See also:

- `learn atom`
- learn-chemical
- `show parray` # to see the create model
- `show` *modelName* # to see the model details
- `Parray(` model *s_name*) : create an empty model/collection for APF or docking or any other types of models.

Atom based predictors

Models to predict single atom properties. (e.g: pKa)

`learn atom` *chemarray* { *R_learnValues*|*RR_learnValues* } `print=`*I_fingerprintParams* `map=`*S_atomMappingParams* [`exclude=`*r_correctionThreshold* [`number=`*i_nofCorrectionIterations*]]

Link or assign reaction group arrays to a Rx positions on a chemical scaffold.

`link group` *scaffold i_R_GroupNumber1 chem_array1 i_R_GroupNumber2 chem_array2 ..*

associate corresponding **R***n* positions on a scaffold with the chemical arrays. This operation copies the *chem_array* into the scaffold, therefore the external array which was used to by this command will remain in place. After the link operation the external array can be deleted.

`link group` *scaffold i_R_GroupNumber* `delete`

delete the association.

`link group` *scaffold i_R_GroupNumber* `table`
extract the R-group associated with the given position from the scaffold into a stand-alone shell chemical table. One can read an RG file with a scaffold and RGroups and a scaffold with linked, but hidden, arrays will be created. Then these arrays can be turned into external chemical tables, edited and `link`ed back to the scaffold.

```
write table mol scaffold "markush.mol" # creates an RG file
#
read table mol "markush.mol"
enumerate library scaffold.mol
```

link internal variables of molecular object

`link` *vs_varChainToBeLinked*

`link molecule` *vs_inSeveralIdenticalMolecules*
impose a chain of equality constraints (*v[1] = v[2] = v[3] = ... = v[n]*) on the specified variables (or, in other words, keep the specified variables equal to each other). If one of the variables is changed all the others will be changed. Energy derivatives are modified accordingly. This command is great for modeling periodic structures (e.g. (Pro-Glu)n).

With option `molecule` , multiple chains of equivalent variables in several molecules will be formed. Make sure that the variables are properly aligned and torsion angles are not linked to phase angles.

Examples:

```
# single chain
 build string "ala ala ala ala ala ala ala ala ala ala"  # 10 alanines
 link v_//phi                   # all the phi angles should be equal
 link v_//psi                   # all the psi angles should be equal
 montecarlo v_/2/phi,PSI v_*    # sample just one residue

# multiple chains for a dimer
delete a_*.
build string "leu ala ala leu ala ala ala leu"
```

```
copy a_ "b"
mv a_2. a_1.
ds a_
set v_1//tvt1 0.
set v_2//fvt1 180.
fix v_//tvt1,fvt1  # do not link those
link v_//* molecule
montecarlo v_1/* v_/*
```

Be careful with selections of `psi` variables in peptides since they are assigned in ICM to the first atom of the *next* residue. `PSI` specification goes around that attribution.

Groups of linked variables can be deleted with the

```
delete link variable
```

command.

Link chains/molecules to sequences and alignments

`link` *ms* [*ali1* .. [`only`] | `alignment` | `sequence` | *seq1 seq2 ... seqN*]
link or associate protein molecules with separate sequences or sequences grouped in an *alignment.* If alignment *ali_* is given, molecules are also linked to this alignment (note that the same sequence can be involved in several different alignments). Amino-acid sequences of amino- or nucleotide-chains in molecular selection *ms_* will be compared with specified ICM-shell sequences and identical pairs will be linked. Make sure that you specify **one** molecule selection, use logical or (|) between the two selections if necessary. Linking molecules with alignments allows an automatic residue-residue assignment by the following commands and functions: `superimpose`, `set tether`, `Rmsd` and `Srmsd` . Alignments can be prepared in advance either automatically by the `align` command or `Align` function, and/or modified by manual editing of the alignment file.

Arguments and options

- *ms_* : selection of chains to be linked, for example `a_*.` that means all molecules of all objects. If no other arugments is specified ICM will rely on the linked sequences to find the latest alignments containing those sequences.
- `only` : the sequences linking is not changed but the link to the specified alignment(s) is attempted
- `alignment` : same as option *ali* .. `only` , but all alignments in the shell will be tried
- `sequence` : ICM will try to link the specified chains (*ms_*) to the sequences in the shell. Name matching will be attempted (e.g. `a_1crn.a 1crn_a`) .
- *seq1 seq2* .. : try to link selected molecules with specified sequences. Note that the sequence should be idential, usually it means that the sequence was produced with `make sequence` *ms_* .

Short forms of the command:

- `link a_*.` # try to find latest alignments for all the chains in the shell
- `link a_*.` ali_target # find the matching sequences leading to the specified alingment, establish links
- `link a_*. sequence` # search all sequences

Use the `ribbonColorStyle="reliability"` option and `color ribbon` to display the local strength of the alignment. The strength parameter will be 3D averaged with the `selectSphereRadius` radius.
The following illustrates the first step of homology modeling.
Example:

```
build "newseq"               # that is what you want to build by homology
read pdb "template.pdb"      # that is the known pdb-template
read alignment "seq3Dali.ali" # prepared/modified sequence alignment
                             # of the two structures
set object a_1.              # this is the first molecule that we
                             # are going to model
link a_*. seq3Dali           # establish links between sequences
                             # and objects
set tether a_1,2.1 seq3Dali  # impose tethers according to the alignment
minimize tether              # fold it according to the template
```

See also:

- ♦ l_autoLink
- ♦ Name (ms alignment)
- ♦ Name(ms sequence)
- ♦ show link *ms*
- ♦ delete link *ms* [alignment]
- ♦ selecting by alignment conservation code (e.g. a_/CX)

list

list [alignment] [command] [factor] [function] [grob] [iarray] [integer] [logical] [macro] [map] [matrix] [object] [profile] [rarray] [sarray] [sequence] [string] [*name1 name2 ...*]

list find *pattern|word*
list ICM-shell objects matching the name pattern (all if name-pattern is omitted). The plural form can be used for more natural expressions. 'list commands' actually means list all legal words known by ICM (ICM command words). Use flanking asterisks to search in any position. Option find or pattern automaticall transforms unquoted word into "*word*"
Examples:

```
 list                          # list the "most wanted" object-types
 list functions
 list sequences                # if you have aliases, you can
                               # type 'ls se' instead
 list "*my*"                   # all ICM-shell variables containing "my"
 list find my                  # the same as the previous search
 list pattern my               # identical to the previous too
```

list graphic font : listing existing fonts for 2D and 3D graphics labels

list font graphic

show currently active fonts used in 2D and 3D labels in the GL graphics window. The output shows the font number, font size, bold-italic-underline and the number of labels using this font. The font **number** refers to the following fonts:

1. courier
2. times
3. arial
4. symbol

Example:

```
list graphic font
 -#-F-sz-biu-rf
 1 2 24      1
```

list the content of the icm binary file

list binary [*s_binaryFileName*]
list the table of contents of the icm-binary multi-object file. The default name is -"icm.icb" and the default extension is ".icb" . To read the whole archive, use the read binary command. For a subset of objects, add the name= *S_listOfNames* option.
Note that the archive can also store graphical view parameters, tethers between the objects, and a string buffer with the last session.
Example:

```
list binary s_icmhome + "example_docking"
 Binary file version: 1
    1 mn_saveAll                     integer                   4
    2 a                              integer                   4
    3                  sarray                   28
    4                     sarray                   28
    5             sarray             396
    6               grob            100992
    7               grob             88596
    8 m_gb                             map                 114280
    9 m_gs                             map                 114280
   10 biotin                           object                5490
   11 DOCK1_rec                        object              265167
   12 displayView                      graphical view         293
   13                 string                   5322
```

```
read binary name={"biotin","DOCK1_rec"} "example_docking"
```

list available sequence databases

`list database`
gives a list of BLAST databases which can be used by the `find database` command for fast sequence database searches. Normally, your system administrator should update the `BLAST` sequence files. ICM just needs a path to this directory which is defined by the $BLASTDB system variable. The output of the command is saved in the `S_out` array. This array can further be processed with the `Field` function.
Example:

```
 list database
 dblist = Field(S_out, 2) # sarray contains search databases
 show dblist
 a=Sequence("PDPPLELAVEVKQPEDRKPYLWIKWSP")
 find database a dblist[2]
```

Trouble-shooting: If you get an error message, check the following:

- check if you have a directory with the blast-formatted files.
- make sure that your `s_dbDir` variable is defined in your `_startup` file and it contains the path to this directory (do not forget the last slash, e.g. `/data/blast/dbf/`). You can always assign it manually from the command line.

---

list directory

See: `Sarray(` *s_filename_filter* directory [`all`]), `sys`

list molcart

`list molcart` [`database=s_dbname`]

gives a list of tables in the molcart database.

See also: `molcart`, `rename molcart`

`list molcart connect`

lists all registered database connections. Note that they do not have to be connected to be listed.

load

load things from the program memory (to load from disk files use `read` command). The opposite action to load is `store`.

load conformation from stack

`load conf` [*os1*] *i_confNumber* [`sstructure`]
assign the *i_confNumber-th* conformation from the `conformational stack` and to the current object (e.g. when you browse conformations accumulated after a `montecarlo` run). If *i_confNumber* is zero, the best energy conformation will be loaded. Montecarlo stack conformations are sorted according to energy values, however you may create your stack manually with an arbitrary order.
Option `sstructure` will automatically recalculate the secondary structure according to the

hydrogen bonding pattens.
Note that the full energy of this conformation which had been stored in the `stack` can be accessed by the `Energy("func")` function.
If an object *os1* is specified, the conformation is loaded from the stack stored in the specified object. The command will update the information about the current conformation in the object's stack.

Example:

```
read stack "f1"            # read conformational stack
load conf 0                # set molecule into the best energy conformation
display a_//ca,c,n         # display the backbone
for i=1,Nof(conf)          # go through all the conformations
  load conf i              # load them one by one
  print Energy("func")     # extract its energy
  pause                    # wait for RETURN
endfor
```

load trajectory frame conformation

`load frame` *i_trjFrameNumber* [*s_trjFileName*] [`sstructure`]
load specified frame from the trajectory. Note that the full energy of this conformation which had been stored by the simulation procedure can be accessed by the `Energy("func")` function.
Option `sstructure` will automatically recalculate the secondary structure according to the hydrogen bonding pattens.
Examples:

```
build "alpha"     # build extended chain of the Baldwin peptide
read trajectory "alpha"
display trajectory "alpha" center # a-ha! conf in frame 541 is interesting
load frame 541 "f1"          # extract conformation from frame 541
print Energy("func")         # print its energy without recalculating
```

create database table view

`load molcart table`[=*s_sql_table*] {*T*|`name`=*s_result_table*} [`filter`=*s_filter*] [`sort`=*s_sort_columns*] [`number`=*i_limit(1000)>] [<connection_options*]

Loads rows from a database table *s_sql_table.* First *i_limit* rows sorted by *s_sort_columns* according to conditions specified in the *s_filter* condition are loaded. If some options are not provided in the command and the table is specified as, the following fields from the table header may be used:

- ♦ `queryLimit` for *i_limit*
- ♦ `queryOrder` for *s_sort_columns*
- ♦ `queryFilter` for *s_filter*

`load molcart table` *T* `refresh` [*connection_options*]

Reloads requested rows from the database based on the ID values in the primary keys column in .

Connection may be specified by *`connection_options`* . If the connection or the table are not specified, this command tries to get their specification from the table header:

- ♦ `querySource` specifies the database table in *database.table* format
- ♦ `queryConnection` describes the connection (not by connectionID)

Tables produced by the **load molcart** command are treated as special "database view" tables in the GUI.

See also: `molcart`, `find molcart`, `query molcart`

load a structural alignment solution

`load solution` [*i_solutionNumber*]
loads the specified solution previously stored by the `align` *`rs_residue1 rs_residue2`* .. command. The two output selections as_out and as2_out contain equivalent residues of the

specified solution. The second object will be superimposed according to the Ca atoms of the found equivalent residues.
Example:

```
 read pdb "4fxc"
 read pdb "1ubq"
 display a_*.//ca,c,n
 color molecule a_*.
 align a_1.1 a_2.1 12 1.5 .1
 center
 load solution 2          # load the second best solution
 color red  as_out
 color blue as2_out
 for i=1,10
   load solution i
   color molecule a_*.
   color red  as_out
   color blue as2_out
   pause                  # rotate and hit 'return'

 endfor
```

load conformational stack from an object

`load stack` *os*

extracts the compressed stack from inside the object and overwrites the existing `stack` . This mechanism can be used to switch between several objects withing one session and use their stacks without any need to work with stack files. Alternatively, in-object-stacks can be saved *with* the object and read back to a session.

Example:

```
  read object "objeWithStack.ob"
  if(Exist(a_ stack)) load stack a_
```

See also:

- ♦ `store stack object`
- ♦ `delete stack object`
- ♦ `montecarlo` .. `store`
- ♦ `set object` .. `stack`
- ♦ `Exist` (*os1* `stack`)

load object from parray or parray in a collection

`load object` *objArr* [`name=`*s*] [`delete`]

Example:

```
read pdb "1crn"
read pdb "2cpk"
p = Parray(object)
delete a_.
load object p[2]
#
c = Collection()
c["ob"]=a_1.
delete a_*.
load object c["ob"] name="x"
load object c["ob"] name="x" delete # overwrite the previous a_x.
```

ICM-shell macros and functions

a named group of ICM commands with arguments that can be called and executed from the ICM shell. A very similar entity is a user-defined (or icm-shell) function that is like a macro but may return a value and be nested. Macros of functions can be :

- ♦ defined or loaded
- ♦ called/executed

Macro can call another macro (nested macros).
Syntax of the macro definition:
`macro` `macroName` [`mute`|`auto`] `prefix1_macroArg1` [(`default1`)] `prefix2_macroArg2` [(`default2`)] ...

icm_commands

`endmacro`

To invoke macro just type its name and provide arguments if necessary.

Naming of the formal arguments of macros and functions. The formal arguments in macros and functions need to be named in a special way to imply the type definition. For example a formal variable which is meant to be a string need to be called `s` or `s_inputstring`

Example of a simple macro without arguments:

```
macro creates_a
  # commands
  a=1
  keep a   # used to push 'a' to the upper level, 'keep global' for all levels.
endmacro
creates_a  # calling macro a
show a     # checking if it creates variable 'a'
```

Example of a simple macro with arguments:

```
macro countMetalNeighbors as_ r_dist (5.)
  l_commands = l_info = no
  nMetals = Nof( Mol( Sphere( as_ , a_*.M , r_dist ) ))
  print " nMetals = ", nMetals
  keep nMetals
endmacro
read pdb "1are"
countMetalNeighbors a_/his,as* 4.
```

Example of a shell-function (not to be confused with the built-in functions):

```
function Bold( s ); return "<b>"+s+"</b>"; endfunction
Bold("Hey") # returns "<b>Hey</b>"
```

See also `function` .

Formal arguments The formal argument names should have explicit prefixes (i_ , r_ , s_ , l_ , p_ , I_ , R_ , S_ , M_ , seq_ , prf_ , ali_ , m_ , g_ , sf_ , as_ , rs_ , ms_ , os_ , vs_ , see `above`) to specify their type. The simplest formal argument name is the prefix without the trailing underscore. If your argument list is incomplete, you will be prompted for the missing argument. Type `q` or enter **empty string** to `quit` the macro without execution. The following features make ICM macros extremely convenient to use:

- no need to explicitly define types of arguments (implicit definition by name)
- one may specify an arbitrary subset of arguments and in arbitrary order if the arguments have different types
- an easy and flexible way to provide defaults in parenthesis after the argument
- automatic prompting of the missing arguments by default, or substituting the default values if the macro is defined as `auto` .
- automatic restoration of all the changed standard ICM-shell variables upon execution.
- new variables defined in the macro are local and will be automatically deleted upon execution, unless they are protected with the `keep` (or `keep global`) command.

Defaults can be provided in parentheses as simple constants (i.e. i_window (8)), or as the whole expressions (i.e. i_1 (mncalls) r_a (Sin(*2.)) i_2) . Default expressions can also be omitted.
Options

- `auto` automatically use defaults for the arguments missing in the command string. Example: **nice "2ins"**. Since the second logical argument l_wormStyle is missing its default value `no` will be used automatically.
- `mute` will suppress automatic prompting. Do not use parenthesized defaults with this option.

The predefined standard ICM-shell `integer`, `real`, and `logical` variables, as well as `preferences` (i.e. i_out, l_warn, wireStyle, PLOT.logo etc.) are be automatically restored upon completion, if changed in the macro, to retain the new value use the `keep` command. Note that the string variables should be restored explicitly. Many `macros` are supplied with the program. Examples:

```
# display molecule as a worm colored from N- to C- term.
 macro dsWorm ms_ (a_*) r_wormRadius (0.9)
  GRAPHICS.ribbonWorm = yes
  GRAPHICS.wormRadius= r_wormRadius
  display ms_ ribbon only
  for i=1,Nof(ms_)           # color each molecule separately
    color Res(ms_[i]) Count(Nof(Res(ms_[i]))) ribbon
  endfor
 endmacro
```

To invoke the macro, type

```
 read object s_icmhome+"crn"
 dsWorm a_1 0.7
```

or just

```
 dsWorm  # and press Enter
```

A set of `ICM macros` is given in the _`macro` file.

See also `function`

---

make

is a family of commands which create new objects of parts of them.

make background : spawning background jobs and processing their output.

`make background` *s_external cmds* [`command=`*s_icmCmdsUponCompletion*] [`info=`*s_Message*] [`name=`*s_jobName*] [`simple`]

This command runs a set of external commands written in a form that can be executed as an external process. Upon execution of these external commands the ICM client will receive the *s_Message* (by default it will be the following message: `"background job 'jobName' completed. Press OK to load the results"`. You can also specify which commands can be executed by the ICM client to load the results of this job. Arguments:

- *s_externalCommands* # e.g. "grep a *.tx >! b" use Path() function to run an ICM thread
- `name=` *s_jobName* # e.g. name= "j1"
- `command=` *s_\n_separated_list_of_ICM_commands* # default is empty string.
- `info=` *s_completionMessage* # e.g. "Finished. Press OK to read the model"

The `make background` command the following features:

- it is portable and works under different operating systems.
- it needs ICM in the GUI mode (`icm -g`)
- to specify a correct external ICM call, you can use the Path(unix, s_options), e.g. `make background Path(unix "_action ") command="read object OUTPUT\nread stack OUTPUT"`

`simple` option creates completely detached job. ICM will not keep any information about that process. This option is useful if you want simply to launch an external program and don't want to have any further interaction with it.

```
make background "ls > tmp.txt" command="read string \"tmp.txt\" " info=""
  #  the empty info arg suppressed the dialog
show s_out
#
make background name="job1" Path(unix,"_myScript",{"-n","-s"})
   # Path(unix,..) returns current ICM location
```

A Windows example:

```
make background "\"c:\\Program Files\\Microsoft Office\\WINWORD.EXE\" C:\\Temp\\Doc1.dc
```

See also:

- ♦ `sys` command
- ♦ `Unix` function

make bond: forming a covalent bond

`make bond` *as_singleAtom1 as_singleAtom2* [`type=`*i_type*]

adds a covalent bond between two selected atoms in a non-ICM molecular object (e.g. X-ray or NMR pdb-entries) or resets the bond type (for ICM objects use `make bond simple` The command is used to correct erroneous connectivity guessed by the `read pdb` command. This correction makes the molecule displayed in the graphics window look better and is necessary before conversion into an ICM molecular library entry (see `icm.res` or user library files) using the `write library` command. It can also be useful to display a connected Ca-trace. In interactive graphics mode you may type `make bond` and then click two atoms with the Control button pressed.
The `type=` option allows one to set the bond type (*i_type* =`{1|2|3|4}` , for a single (default), double, triple and aromatic bond, respectively.

`make bond simple` *as_singleIcmAtom1 as_singleIcmAtom2*

forms a bond, e.g. for peptide cyclization. One needs to unfix the following energy terms for that bond to be minimized properly:

```
build string "ala ala ala ala ala"
make bond simple a_/1/n a_/5/c
set term "bb,bs,af"
minimize
```

---

make bonds in an atomic chain

`make bond chain` *as_chainOfAtoms*
connects specified atoms in a linear chain. Useful for PDB entries containing only Ca atoms.
Examples:

```
 read pdb "4cro"        # contains only Ps and Ca's
 display                # Milky sausage
 make bond chain a_4cro.//p   # connect P atoms of the DNA backbone
 make bond chain a_4cro.//ca  # connect Ca atoms of the protein backbone
```

See also: `delete bond`.

---

make boundary: Poisson electrostatics

a command to prepare for the boundary element electrostatic calculation
`make boundary` [*as*]
this is an auxiliary command which is required if you need to calculate the electrostatic free energy with the `boundary element` method several times. Optional atom selection `as_` from which the electrostatic field is calculated can be specified. This may be the case if the charge distribution changes but the shape does not. However, the boundary does depend on the dielectric constant parameters such as `dielConst` and `dielConstExtern` . If you intend to change them the boundary need to be remade every time. This command does not generate any output by itself, it just creates the internal table which can later be used by the `show energy` command or the `Potential( )` function.
The dielectric boundary is a smooth analytical surface which is built with the contour-buildup algorithm (`Totrov,Abagyan, 1996`). The surface looks like the `skin` surface, but uses different radii which were optimized against experimental LogP data. Both skin and the dielectric boundary uses the same water radius (the `waterRadius` parameter). The "electrostatic" radii used by ICM to calculate the boundary are stored in the `icm.vwt` file.
See also: `REBEL`, `surfaceAccuracy`, `electroMethod`, `delete boundary`, `show energy",term "el"`, `Potential( ).`
Examples:

```
 electroMethod="boundary element"
 read object s_icmhome+"rinsr"
 delete a_w*                                       # get rid of water molecules
 make boundary a_1 surfaceAccuracy = 5          # calculate params of the dielectric b
 show energy "el"                               # electrostatic energy by BEM
 e1=Energy("el")                                # extract the energy
 set charge a_/33/cd*,hd*,ne*,he*,cz,nh*,hh* 0.  # uncharge arg33
 show energy "el"                               # electrostatic energy of the uncharge
 e2=Energy("el")                                # extract the energy
 print e1-e2
 delete boundary                                # memory cleanup
```

make directory

`make directory` *s_Directory*
make specified directory. Example:

```
make directory "/home/doe/temp/"
```

See also: `sys` , `set directory`, `delete directory`, `Path`(directory)

make disulfide bond

`make disulfide bond` [`only`] *as_atomSg1 as_atomSg2*
form breakable disulfide bonds between two sets of specified sulfur Sg atoms, regardless of the distance between them. Forming the bond means that two Hg hydrogens of Cys residues are dismissed, a covalent bond between two Sg is declared (but not enforced) and four local distance restraints (see `icm.cnt`) are imposed. These restraints are indeed local, since two Sg atoms only start feeling each other when they are really close, otherwise the energy contribution is close to zero . Option `only` causes deletion of previously formed disulfide bonds, otherwise the new one is added to the existing list of disulfide bonds.

Examples:

```
 build string "se cys ala cys" # sequence containing two cysteins
 display                       # display an extended ICM model of the sequence
                               # set only one SS-bond, disregard all previous
 make disulfide bond a_/1 a_/3 only
 montecarlo                    # MC search for plausible conformations
```

See also: `delete disulfide bond` and **(important!)** `disulfide bond`.

make drestraint: extract distances structure

`make drestraint` *as_select1 as_select2 r_LowerBound r_UpperBound r_LowerCorrection r_UpperCorrection* [*s_fileNameRoot*]
create two files containing the list of all the atom pairs specified by two selections (i.e. a_* a_* - all the pairs; a_1//* a_2//* atom pairs between molecules 1 and 2 for which the interatomic distance lies between *r_LowerBound* and *r_UpperBound.*
Note: it is critical that both selections are in the **same** object. Only `tethers` can pull to atoms of a different object.
For each pair of atoms a `distance restraint type` is created with lower bound less than the actual interatomic distance by *r_LowerCorrection* and upper bound greater than the actual interatomic distance by *r_UpperCorrection.* This command can be used for example to impose loose distance constraints between two subunits.
The number of the formed drestraints is returned in the `i_out` variable.
See also: `set drestraint` as_1 as_2 i_Type
if you want to impose a specific drestraint.
Examples:

```
 read object s_icmhome+"complex"  # load a two molecule complex for refinement
                                  # extract all Ca-Ca pairs between 2 and 5 A
                                  # for each pair at distance D create distance
                                  # restraint type with lower bound D-2.5 and
                                  # upper bound D+2.5
 make drestraint a_1//ca a_2//ca 2. 5. 2.5 2.5
```

make factor: FFT calculation of diffraction amplitudes and phases

make factor *map_Source* {*I_3Maximal_hkl* | *r_resolution*} [*s_factorTableName*[*s_ReName*[*s_ImName*]]]
calculate structure amplitudes and phases from the given electron density map by the Fast Fourier transformation. The table ' *s_factorTableName'* with h,k,l and structure factors will be created (further referred to as **T** for brevity). It will contain the following members:

- three integer arrays of Miller indices: **T.h T.k T.l**
- two rarray of real and imaginary parts of the calculated structure factors. Default names: **T.ac** and **T.bc**, respectively. Alternative names can be explicitly provided in the command line.

If structure factor table *s_factorTableName* already exists, structure factor real and imaginary components are created or updated in place. Any other arrays containing experimental, derivative or control information may be added to the table and participate in selections and sorting.
Example:

```
read map s_icmhome+"crn"      # load "crn.map"
set symmetry m_crn 1
make factor { 5 5 5 } "F"     # h_max=k_max=l_max=5
                              # F.h, F.k, F.l, F.ac, F.bc are created
show F
group table append F Sqrt(F.ac*F.ac + F.bc*F.bc) "Fc" Atan2(F.bc,F.ac) "Ph"
sort F.Fc
show F
```

make flat chem_array

make flat *chem_array* [rotate] [hydrogen] [window=*r_WidthToHeightRatio*] [index=*I_indices*]

convert a chemical array into standard automatically generated 2D chemical drawings *in place* (compare with Chemical(Smiles(*chem_array*), smiles) which does not touch the source array). A chemical array can created by the read table mol command. The compounds in the source file can be 0D (all coordinates set to 0), 3D or 2D. In all cases these x and y coordinates are can not be used for chemical drawing and one needs to use the above command to generate 2D drawings. The command also preforms rotation for optimal fit into rectangle with specified width to height ratio (*window* argument). Default ratio is 1.5.

Other options:

- *rotate* : does not coordinate assignment, preforms only best fit rotation
- *hydrogen* : keep explicitly drawn hydrogens
- *index* : performs operation only on selected compounds

Example:

```
read table mol s_icmhome+"template_3D.sdf"
make flat template_3D.mol
```

See also: Chemical , read table mol .

make grob map command to contour electron density or grid potentials

make grob *m_map* [header] [solid] [box] [*I_indexBox*[1:6]] [[exact] [field=]*r_sigma*|*r_absValue*] [*as* [margin=*r*]] [name=*s*]

make grob m_map add *r_sigmaIncrement* make grob m_map add exact *r_absoluteIncrement* # build a contour that can be modified

make grob g_existingContourGrob add *r_sigmaIncrement* # rebuilds and redraws an existing contour

Create graphics object by contouring electron density map at a given threshold.

threshold: By default the contouring level is calculated as the mean map value (returned by Mean (*m_map*)) plus mapSigmaLevel times root-mean-square deviation value. If a real value argument is provided, the mapSigmaLevel shell variable is redefined. Option exact allows one to specify *absolute* value at the contouring is performed. If atom selection is specified, contour will only be built around *as_*, with the optional additional `margin`. Helpful in contouring ligand from electron density map.

Other options:

- ♦ `header` this option adds the name of the source map and the command to recalculate the grob at different contour level.

Example:

```
build string "his glu"
make map potential Box( a_ 3.)
make grob m_atoms 3. # 3 sigmas above the mean
#  make grob m_atoms .2 exact # countour at 0.2 level
#  .2 or .1 exact is useful to detect almost closed pockets
display g_atoms
#
make grob m_atoms exact 0.15 # at value of 0.15
display g_atoms
#
mapSigmaLevel = 1.5
make grob m_atoms add 0. # at mapSigmaLevel
make grob g_atoms add -0.1 # at 1.4 sigma
#
loadEDS "1atp" 0.
read pdb "1atp"
make grob m_1atp 1.5 a_atp
cool a_
display g_1atp
```

Defaults:

- ♦ create simple chicken wire map (sections in three sets of planes, NOT solid)
- ♦ take the `current map`;
- ♦ generate the name of the grob which is the same as the map name except for the `g_` prefix;
- ♦ contour the whole map
- ♦ use threshold value from the ICM-shell real variable `mapSigmaLevel` .
- ♦ `mapSigmaLevel` is changed if the exact option is used

Option `solid` tells the program to create a solid triangulated surface which can later be displayed by `display grob solid` command. The threshold is expressed in the units of standard deviations from the mean map value, i.e. 1.0 stands for one sigma over the mean. *I_indexBox* [1:6] is optional 6-dimensional `iarray` containing { i_startSection i_startRow i_startColumn i_NofSections i_NofRows i_NofColumns }. It overrides the default, contouring the whole map. Option `box` adds surrounding box to the grob.

---

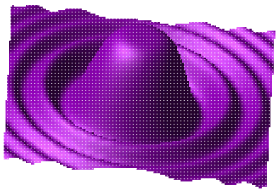
make grob image command to create a vectorized graphics object.

`make grob image` [`name=`*s_grobName*]
create a vectorized `graphics object (grob)` from the displayed wire or solid objects. The information about colors will be inherited. Very useful if you want to export wire, `ribbon` or CPK into another graphics program, since graphics objects can be `written` in portable Wavefront (.off) format. Further, graphics objects can exist independently on the molecules which may be sometimes convenient. Also, underlying lines and vertices can be revealed. The graphics object created from the displayed solid representations assigns and retains color information as lit in a given projection. These colors can not be changed. Use special `make grob skin` command to generate a more elaborate graphics object from `skin` .
Examples:

```
read object s_icmhome+"crn"
ds a_crn.//!h* ribbon          # ribbon
make grob image name="g_rib"
display g_rib smooth only      # try select g_rib and Ctrl-X,Ctrl-E/W etc.
                               # option smooth eliminates the jaggies.
write g_rib                    # save to a file
```

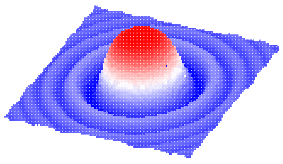
---

make grob matrix



make grob [solid] [bar[box]] [color] *M_matrixName* [*r_istep r_jstep r_kstep*] [[name=]*s_grobName*]
Create a three-dimensional plot from *M_matrixName*, so that x=i* *r_istep*, y=j* *r_jstep* and F(x,y)= k* *M_matrixName[i*,j]. Options:

- bar : generate rectangular bars for each i,j matrix value instead of a smooth surface.
- box : add a box around the 3D histogram
- color : color grob by value according to the PLOT.rainbowStyle preference.
- solid : tells the program to triangulate the surface



Examples:

```
 read matrix s_icmhome+"def"
 make grob def solid
 display
# OR
 read matrix s_icmhome+"ram"                 # phi-psi energy surface
 make grob ram 1. 1. 0.1                     # create the surface
 display g_ram magenta                       # display it
 make grob solid ram 1. 1. 0.08 name="g"     # create the surface
 display g solid gold                        # display it
```

make grob potential

make grob potential [solid] [*as_1* [*as_2*]] [[field=]*r_potentialLevel*] [grid=*r_gridCellSize*] [margin=*r_margin*] [[name=]*s_contourGrobName*]

Example:

```
make grob potential a_lig
```

create graphics object of isopotential contours of electrostatic potential which takes not only the point charges but also the dielectric surface charges resulting from polarization of the solvent. This potential need to be calculated in advance by the boundary element algorithm. Contours can be displayed in the wire and solid representations (see also display grob). The default parameters are:

- *r_polentialLevel* 0. kcal/mole/electron_charge_units.
- *r_gridCellSize* 0.5 A (you may want to increase it up to 2A for speed).
- *r_margin* 5.0 A (you may want to reduce it for speed).

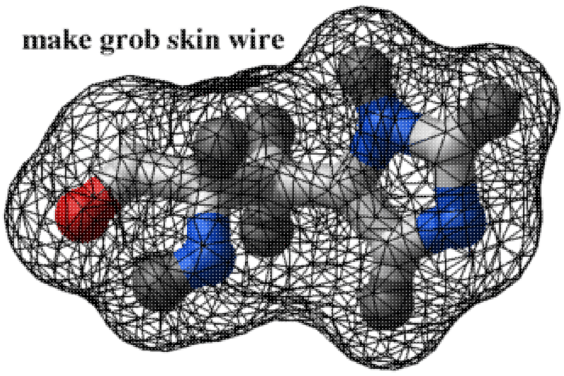
See also: make map potential, electroMethod, make boundary, show energy "el", term "el", Potential( ).
Examples:

```
 build string "se his arg glu"
 electroMethod="boundary element" # REBEL algorithm
 make boundary
 make grob potential solid 0.1 grid=2. margin=4. name="g_equipot1"
 display g_equipot1 transparent blue
 make grob potential solid field=-0.1 grid=2. margin=4. name="g_equipot2"
 display g_equipot2 transparent red
 ds xstick residue label
```

make grob skin or surface

`make grob skin` [`wire` | `smooth`] [*as_1* [*as_2*]] [[`name=`]*s_grobName*] [`r_transparency`]

`make grob surface` [`color`] [`wire` | `smooth`] [*as_1* [*as_2*]] [[`name=`]*s_grobName*] [`r_transparency`]



create `grob` containing the specified `molecular surface` (referred to as skin). If the `wire` option is given the transparent wire grob will be created (solid grob is the default). It will have the same default color. The disconnected parts of this grob may later be `split` . The grob will be named by the default name g_*objName* unless the name is explicitly specified. The final **actual name** will be returned in `s_out` .
The `smooth` option allows one to close the cusps. This closure is necessary to enable the `compress` grob operation. The `compress` *g_* command allows one to dramatically simplify the triangulated surface and reduce the number of triangles. Typically `compress g_ 1.` will reduce the number of triangles by an order of magnitude.

A grob can later be colored with the `color grob potential` command.
Examples:

```
read object s_icmhome+"crn"
           # skin around a substructure, (just as an example)
make grob skin a_/1:44 a_/1:44 0.6
split g_crn_m
display g_crn_m2  a_//*
show Area(g_crn_m2), Abs(Volume(g_crn_m2))

make grob skin a_ a_ name="gg1" # display gg1 now
make grob skin wire  name="gg2" # display gg2 now

make grob skin smooth a_/1:20 a_/1:20  name="gg3"
compress gg3 1. # simplifies the surface
```

The transparency can also be set with the `set grobname r_transparencyLevel` command.

See also: `set color` to set atom colors

Creating 3D label objects

A number of commands in ICM enable the creation of "3D label" objects which help to measure and annotate geometry in the 3D space, like distances and angles. Some 3D labels, like hydriogen bonds, illustrate concepts which depend on the geometry and the structure of molecules. 3D labels are stored in a `parray` object of a "label3d" subtype.

3D labels defined on atoms are dynamic: visual angle/distance information is updated depending on the changes in the atom geometry.

3D label creation commands have similar structure. Commands which are currently available are:

- ♦ `make distance` to create distance labels;
- ♦ `make hbond` for hydrogen bonds;
- ♦ `make angle` for planar angles;
- ♦ `make torsion` for dihedral angles.

Each of these commands has specific arguments. but there is a number of common options:

`name` = *s_n*	the name of the created array of 3D labels
`append`	append new 3D labels to the existing array;
`delete`	forcefully delete the array before repopulating it with 3D labels;
`refresh`	keep all existing 3D labels which are outside the specified selection, rebuild 3D labels inside the selection;
`display`	display the created labels.

make distance

`make distance` *as_1* [*as_2*|`auto`] [`molecule`|`align`] [*r_maxdist*] [*3d label options*]

creates a `parray` with distances between all atoms in *as_1* and *as_2*, or all atoms within *as_1* if the second selection is not specified.

Output

- the distance `parray`
- `i_out` with the number of distances in that parray

Examples:

```
read object s_icmhome+"crn"
display ribbon wire
make distance a_//oe* a_//ne,nh* display
make distance a_//oe* a_//ne,nh* 5. delete display
```

Example in which we detect clashes:

```
 make distance a_//!vt* a_//!vt* 1.5 # creates a distance object
 CLASHES = Table( $s_out distance )  # s_out stores name of distance-object
 sort CLASHES.dist
 show CLASHES.dist<0.8
```

See also:

- `make 3d label` and `make hbond` for further comments on the command options.
- `Table(distobj distance)`
- `Nof(distobj distance)`

make hbond

`make hbond` *as_1* [*as_2* | `auto`] [`molecule` | `align`] [*r_maxdist*] [*3d label options*]

creates an array with all hydrogen bonds contained in *as_1*, or between *as_1* and *as_2* if the second selection is specified. Hydrogen bonds are calculated according to several shell parameters listed below. It is possible to specify the upper limit for the distances between atoms which will be considered as potential bond using the *r_maxdist* value.

Other options and shell parameters:

- `molecule` option forces to create only intermolecular bonds
- `auto` mode automatically sets the `molecule` mode if contains more than one molecule
- `align` : "1-1 selections"
- `GRAPHICS.hbondAngleSharpness` (1.7)
- `GRAPHICS.hbondBallPeriod` (1.2)
- `GRAPHICS.hbondMinStrength` (1. , allowed range (0.,2.))
- `GRAPHICS.hbondWidth` (0.6)
- `GRAPHICS.hbondBallStyle`
- `GRAPHICS.hbondRebuild`
- `GRAPHICS.hbondStyle` (label style)

Examples:

```
read object s_icmhome+"crn"
display ribbon wire
make hbond a_ display GRAPHICS.hbondMinStrengh=0.5 name="x"
t = Table(x,distance)
sort t.dist
show t[1]
```

See also:

- `make 3d label` for the explanation of 3d label options
- `"dynamic" single object hydrogen bonds`
- `Table(hbonddist distance)`

make angle

`make angle` *as_3_connecting_atoms* [*3d label options*]

creates a planar angle 3d label. Requires a selection containing 3 connecting atoms.

See also: `make 3d label` , `Table(angles,distance)`

make torsion

`make torsion` *as_4_connecting_atoms* [*3d label options*]

creates a dihedral angle 3d label. Requires a selection containing 4 connecting atoms. Examples:

```
 read object s_icmhome+"crn"
 display ribbon wire
 make angle a_/22/c | a_/23/n,ca display
 make torsion a_/22/ca,c | a_/23/n,ca display
```

See also: `make 3d label` , `Table(tors, distance)` .

make/store graphical image to image parray

`make image` [`library=`*s_albumName*] `name="Party2007.png"`

will save the graphical image to the internal album (a `parray`). The name will be used if you decide to save the album to a file.

Example:

```
make image  # will create album if it does not exist
make image  #
make image  #
delete variable album 2
```

make key # obsolete

`make key` {*s_smiles* | *as*} [*S_arrayOfFragmentSmiles*]
Note: the `make key` command is obsolete. The new chemical fingerprints are dynamic and does not have a predefined set of fingerprints of variable length. The new fingerprints are used in the following commands and functions:

- `Distance(` *chem1 chem2* `)` or `chemical distance matrix`
- `search in Molcart databases`
- `search in loaded 3D objects`
- etc.

This command generates a binary chemical key, i.e. a bit-string in which each bit corresponds to a chemical substructure, converts the bit-mask into the hexadecimal string and saves this hex-string in `s_out` . The bit-string with chemical substructure information can then be used to calculate the Tanimoto similarity distance with another chemical key.
By default the `make key` command uses a built-in array of 96 substructures, and generates a 24-character hex-string (each hex-character codes for 4 bits), however any string array of subfragment smile-strings (*S_arrayOfFragmentSmiles*) can be provided.
The hex-string can be converted back into an array of bits packed into integers with the `Iarray(` { s_chemkey | S_chemkey } `key`) function.
The bit-distance matrix between two arrays of bit-strings represented by two iarrays can be calculated with the `Distance(` `Iarray` (*S_1* key) `Iarray` (*S_2* key) *i_nBits* [`key`]) or `Distance(` `Iarray` (*S_1* key) `Iarray` (*S_2* key) *i_nBits* `simple`) functions, where the number of bits, *i_nBits*, is usually `96`, unless you use a user defined array of fragments. There is also a weighted form of the chemical key distance (see the `Distance` function). By default, or with the `key` option, the function returns matrix with the `Tanimoto similarity distance` (0. all bits are the same, 1. no bits in common), while with the `simple` option the second chemical key is considered as a sub-fragment and the distance becomes `0.` (identity) if the

sub-fragment is present in the first bit-mask.
Examples:

```
  read mol s_icmhome+"ex_mol.mol"
  S_key = Sarray()
  for i = 1, Nof(a_*.)
    set obj a_$i.
    build hydrogen
    set type mmff
    convert
    smil = Smiles(a_)
    make key smil
    S_key = S_key // s_out
    delete a_
  endfor
  S_key
```

---

make map

A family of commands producing maps. It includes:

- `make map cell`
- `make map potential`
- `make map factor`

make map cell

`make map cell` *R_6cellParameters I_3NofSteps* [*R_6box* | *I_6box*] ["`zxy`"] [*as*] [`name=`*s_mapName*]
create an electron density distribution for atom selection `as_` (all atoms of the current object by default) on a three-dimensional grid. See also `make map potential` for a rough electron density map. The electron density is calculated from the cartesian coordinates of the selected atoms using a 2-Gaussian approximation. If the `l_xrUseHydrogen` logical is set to `no` , hydrogen atoms are ignored. The following parameters are taken into account:

- the shape of the Gaussian is influenced by the individual atomic b-factors (see `set bfactor`).
- `addBfactor` is added to individual atomic B-factors

R_6cellParameters is a real array containing { *a b c alpha beta gamma* } parameters. Optional *R_6box* or *I_6box* arrays define the corner of the map box (closest to the origin) and its sizes ({ *x1 y1 z1 dx dy dz* } or { *nx ny nz dnx dny dnz* }, respectively). The whole cell is taken by default.
Examples:

```
 read object s_icmhome+"crn"
 make map cell {5. 5. 5. 90. 90. 90.} 0.5 a_//ca,c,n
```

---

make map factor : calculate electron density map from structure factors

`make map factor` [*T_factor*] [m_map]
calculate an electron density distribution on a three-dimensional grid from a structure `factor` table of the Miller indices, reflection amplitudes and phases. Requires that the map is created before with the `make map` command. If optional arguments are not given the `current map` and/or `current factor` will be used. A new empty map can be created from an empty selection by the

```
 make map a_!*
```

parameters # see the `make map cell` command.

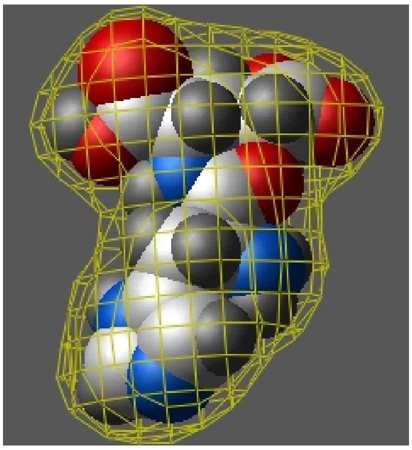
---

make map potential: grid energies, converting crystallographic electron density maps

`make` `map` `potential` [`simple`|`occupancy`] [*s_terms* | `name=`*s_mapname*] [*as*] {[*R_6box*] [*r_gridCellSize*] | `cell=`*map* } [`l_ecep=`no|yes]
create a property `map` for the *as_* selection. This command is used for low-resolution surface generation or to make grid potential maps for fast docking. The optional arguments are the following:

- *s_terms* : a smooth Gaussian atom density map is generated by default, otherwise the grid energy maps specified by the 2-letter terms are calculated, e.g. `gc,gh,gs,ge,gp`). The names of the generated maps are standard and can not be changed.
- *as_* selection : All atoms of the current object are taken by default.
- *r_gridCellSize* : by default is 0.5 A for small objects, the default increases with the size of the object. We do not recommend to use values over 7 A for very large objects.
- *R_6box* : default it is a box around the selected atoms plus 3A margins. The box defines coordinates of the two opposite corners of a box (see also the `Box` function).
- Option `cell` = *map* replaces the box and the grid size and uses the parameters from the *map* instead. This option also allows one to use an *oblique* cell.
- flag `l_ecep` = `no` this option affects the "`gb`" hb-donor field calculation (see below). It allows one to project the field *from* the donors (i.e. D-H) and *for* the hbond *acceptors* further away from the donor atom along D-H bond vector to map out a realistic location for a heavy atom acceptor.

The default map (with no terms provided) will return an electron density map. It supports the `occupancy` argument as well.


`m_atoms` contoured at `0.3` `exact` level. The `0.5` level is closer to the van der Waals surface.

Each individual term (say, "gp") may result in creation of one or several (up to seven) different grid maps. These are the maps created under different terms:

- "gh" : `m_gh` (grid for hydrogen probe)
- "gc" : `m_gc` (grid for a standard heavy atom probe, say C,N,O), `m_gl` (grid for large atoms, like metals and halogens)
- "ge" : `m_ge` (grid for electrostatic potential)
- "ge" with Generalized Born (`electroMethod` = 5): `m_ge`
- "gb" : `m_gb` (grid for the hydrogen bonding potential). Combines donor and acceptor fields. `l_ecep` variable affects the donor field.
- "gp" : up to seven grids named as `m_g1`, `m_g2`, `m_g3`, `m_g4`, `m_g5`, `m_g6`, `m_g7`

- default (no terms specified): atomic density map `m_atoms` ; if contoured, `m_atoms` generates a smooth Gaussian envelope around a molecule (see Figures)

- ♦ `simple` : option to enforce a single `m_gc` map instead of the default pair of maps: `m_gc` and `m_gl.`
- ♦ `occupancy` : option to attenuate the map intensity by the atom occupancy.

```
 build string "his arg"
 display cpk
 make map potential Box( a_ 3.)

# wire surface
 make grob m_atoms 0.3 exact  # contours near vw-radius.
 display g_atoms

# solid surface
 make grob m_atoms solid 0.5 exact
 display g_atoms smooth
```

- ♦ term "el", map m_el : Coulomb electrostatic grid, contributions truncated at +-100. kcal/mol.

```
 build string "se his arg"
 make map potential "el"  Box( a_/1,2/* , 3. )
 display a_
 display map m_el {1 2 3}
 make grob m_el exact  # contouring at 0. potential
 display g_el
 set occupancy a_/arg/!ca,c,n,o,cb  0.3
 make map potential simple occupancy "gc"  Box( a_/1,2/* , 3. )
# enforce a single map with attenuated side chain
```

- ♦ term `"gh"` : van der Waals grid for a hydrogen probe, grid potential is truncated from above according to the `GRID.maxVw` parameter;
- ♦ term `"gc"`, map m_gc : van der Waals grid for a carbon probe; grid potential is truncated from above according to the `GRID.maxVw` parameter; By default, terms `"gc"` will generate **two** maps: `m_gc` and `m_gl` , the van der Waals map for large atoms with van der Waals radius larger than 1.8A. To enforce pushing all non-hydrogen atoms to a single `m_gc` map, use the `simple` option.
- ♦ term `"ge"`, map m_ge : electrostatic grid; grid potential is truncated from above and below according to the `GRID.maxEl` and `GRID.minEl` parameters;
- ♦ term `"gb"`, map m_gb : hydrogen bonding grid;
- ♦ term `"sf"`, map m_ga : surface accessibility grid. This map is not an independent term, but allows one to correctly calculate atomic accessible areas if a part of the system is presented by the grid potentials. If a map named `m_ga` is present it will be automatically taken into account in energy calculations of the `"sf"` term.

Fine-tuning the maps Sometimes you want the van der Waals grids, "gh" and "gc", generated from the whole receptor, while the "ge" or "gb" grids generated only from a small region of the receptor. In this case you can run the command two times with different source-atom selection. Example:

```
read object s_icmhome+"crn"
make map potential "gh,gc" a_1 Box(a_1)
make map potential "gb,ge,gs" a_1/15:18,33:46 Box(a_1)
write m_ge m_gc m_ge  # write three maps at once
```

A different method is to use the `Bracket(` *m, R_6box*) function which sets everything beyond the box to zero. Noted that the in the above method only the selected residues make contribution. In the following method all residues make contribution, and then the resultant map is truncated in space. Example:

```
rename m_ge m_ge1  # Compare with the map generated in previous example
make map potential "gh,gc,gb,ge,gs" a_1 Box(a_1)
m_ge = Bracket(m_ge, Box( a_1/15:18,33:46 ))     # redefine m_ge
display m_ge
display m_ge1
```

See also: `make map potential` *`m_electronDensity`* to generate a rectangular grid from an oblique crystallographic density map.

make molcart

make molcart table *T* name=*s_dbtable* [*make_options*] [*connection_options*]

Imports data from ICM table *T_* into database table *s_dbtable.*

make molcart table *s_filename* [mol | smiles | separator [header]]
name=*s_dbtable* [*make_options*] [*connection_options*]

Imports data from the *s_filename* file into database table *s_dbtable.* The file format may be guessed from the *s_filename* or specified explicitly. Supported formats are:

- ♦ mol for SDF
- ♦ smiles for SMILES
- ♦ separator for CSV

For CSV files the header directive tells to treat the first row in the file as column names

Connection may be specified by *connection_options* .

Other options (*make_options*) available:

- ♦ append : append to an existing database table (as oppposed to overwriting it)
- ♦ column= *S_column_specs* : allows one to specify requested columns and some other properties of them in a special format.
    - ◊ unique
    - ◊ fulltext
    - ◊

make molcart table input=<[s_connectionID;][s_db.]s_sourceTable>
name=<[s_db].s_targetTable> [append] column=*S_column_specs connection_options*

Copies data from one molcart table to another.

- ♦ input : string which contains source table table name (from which data will be copied). Optionally you may add connection ID and database name.
- ♦ name : string which contains target table and database (optionally) names.
- ♦ append : append to an exiting table. If table does not exist it will be created.
- ♦ column : sarray with column names to be copied. By default overlapping subset of column names between two tables will be used.

Example:

```
add column t Chemical({"CCC","CCO"})  # create a local table
make molcart table t name="test.t"    # copy it to molcart
make molcart table input="test.t" name="test.tt"  # make another copy
make molcart table input="test.t" name="test.tt" append  # append the same table again
print  Nof( "test.t" sql ),  Nof( "test.tt" sql )  Nof( "test.tt" molcart unique )
```

See also: molcart, write molcart.

SAR analysis

make molsar {*T_chemicalTable*|*X_chemicalArray*} *X_scaffold* [auto] [append]
[name=s_ResultTable>]

Performs R-group decomposition using X_scaffold. This command is similar to split group command. The only difference is that single table will be generates. For each R-group in the X_scaffold a column will be created.

With append option the original compound will be added to the result table.

With auto option no explicit R-group specification is needed. The command will automatically find attachment positions and create appropriate columns. Columns which are invariant (no changes in substituent) will be excluded.

Example:

```
group table t Chemical( { "C1NC(C(C1O)O)CO","C1NC(C(C1O)C)CO" } )
make molsar t Chemical("C(NCC1)C1") auto name="tsar1"
make molsar t Chemical("C(NC(C1)[R1])C1[R2]") auto name="tsar2"
```

See also: `split group` , split-groupenumerate-library , `make reaction` , `Replace chemical` , `Find chemical`

split molsar: generate SAR table from the result of `make-molsar

`split molsar` {*T_chemRgroupTable|R1_col R2_col*} [*R_propColumn1 R_propColumn2 ...*] [`name=`*s_sarTable*] [`color=`*R_*] [`rainbow=`*s_rainbow*]

generates one or more 2D SAR analysis tables from the input R-group columns.

```
group table t Chemical( { "C1NC(C(C1O)O)CO","C1NC(C(C1O)C)CO" } )
make molsar t Chemical("C(NC(C1)[R1])C1[R2]") auto name="tsar2" append
split molsar tsar2.R1 tsar2.R2
```

make pca

`make pca` [append] *T* [*i_nPC=3*] [*r_tradeoff=0.010000*] [*options*]

Options:

- `name=` *s_prefixForPCcolumnNames*, e.g. `make pca t name="pc"` will create columns `pc1` and `pc2`
- `select=` *I_rows* , allows one to specify rows should be analyzed that e.g. `1,2,5,8,12`. Other rows will get zero values.
- `column=` *S_columnNames* , allows one to specify columns for the PCA analysis. The column names can be specified in any of these formats: *"colName"* , *".colname"* , *"tabname.colname"*, e.g. {"A","B","C"} . See also `Name` (e.g. `Name( t number )` for the names of all numerical columns .

make peptide bond

`make peptide bond` *as_C as_N_or_S*
form the peptide bond between two selected C- and N- atoms, or the thioester bond between C- and S- atoms. The bonds may be formed between the terminal amino- and carboxy- groups (a_/1/n and the last c), as well as between such amino acid side-chains groups as a_/lys/nz and a_/asp,asn/cd, a_/glu,gln/cg. The energy restraints for form the additional chemical bond will be calculated under the "ss" term and can be viewed with `show drestraint` See also: `delete peptide bond` `How to modify an ICM-object` .
Example:

```
 build string "se nh3+ gly gly gly gly gly his"  # notice: NO C-term group
 display
 make peptide bond a_/nh3*/n a_/his/c       # form a cyclic peptide
# display drestraint
 minimize "ss"
 minimize "vw,14,hb,el,to,ss"
#
# form thioester bond
#
 build string "se cys ala ala ala glu"
 display
 make peptide bond a_/1/sg a_/5/cd
 minimize "ss"  # term "ss" is responsible for the extra drestraints
```

Note that this method uses distance constraints (as it is done for the disulfide bonds) to support the closure. Another method:

```
make bond simple .. ..
set term "bb,bs,af"
```

will use the force field to keep the peptide closed.

Generate an attractive grid map from crystallographic electron density map for refinement

`make map potential` *map_Xray* [*as*] [*R_6box*] [*r_gridCellSize*] [`smooth`] [`name=`*s*] converting crystallographic density map which is not suitable for energy manipulations to a grid potential `m_xr` . The source map starts at {0.,0.,0.}, can be oblique with uneven spacing. However the resulting map is always equally spaced rectangular map in any area of space. This grid potential can be used in real space refinement. Find the description of the arguments in `make map potential` command.

Option `smooth` uses the Gaussian smoothing of the values.

```
loadEDS "1atp" 0.  # downloads electron density map m_1atp
read pdb "1atp"
make map potential m_1atp a_atp # map around around ATP
m_gc = - m_xr  # a possible use. Also possible with "gp"
set terms "gc" # add gc germ for minimization
```

See also:

- `make map cell`

make plot: Adding a scatter plot or a histogram to a table

`make plot` *T s_plotArgs* [`name=`*s_plotname* [`append`]|[`output=`*s_fileName* [`size={`*w*,*h*`}`]]

This command creates a plot belonging to table *T* . Each table can contain multiple plots and all the plots belong to the following member of the table header: *T*`.plot`

The *s_plotArgs* string contains the arguments, e.g.

```
  x="Random(1.,10.,100)"
  add column t $x $x Shuffle(Sarray(50,"red")//Sarray(50,"blue")) $x $x  name={"x","y",
  make plot t "x=A;y=B;color=C;shape=D;size=E;"  # or
  make plot t "x=A;y=B;color=C;;element=rectangle;center=1.3,2.4;radii=1.3,2.0;color=re
  add header t Matrix(10)
  make plot t "matrix=A;rainbow=white/yellow/green"
```

The syntax of the *s_descr* string containing other arguments of the `make plot` command is the following. *plot_element*[**;;***plot_element2***;;...**][*general_plot_properties*]

Plot elements. Each double-semicolon separated section is either a plot/histogram or a geometrical element.

- XY scatter plot, contains both `x=` and `y=` plus optional `color=` , `rainbow=` and `size=`.
- *histogram* data plot, contains either `x=` OR `y=` , not both of them, plus
- `element=rectangle;center=`*x*`,`*y*`;radii=`*rx*`,`*ry*
- `element=rectangle;x1=`*x1*`;y1=`*y1*`;x2=`*x2*`;y2=`*y2*`;`
- `element=ellipse;center=`*x*`,`*y*`;radii=`*rx*`,`*ry*
- `element=polygon;xy=`*x1*`,`*y1*`,`*x2*`,`*y2*`,..`

Scatter plot arguments

- `x=`*columnName* # must be present
- `y=`*columnName* # must be present
- `color=`*color_or_columnName*;
- (requires `color=`) `rainbow=`*color1*[**/***color2***...**][**,***from:to*] (e.g. `make plot t "x=A;y=B;color=C;rainbow=red/white/blue,100:150,linear/0:0/0.7:0.5`
- `size=`*number or column*;
- `shape=`*shapeName_or_columnName*. The shape names are the following: `shape=Circle|DTriangle|Diamond|Cross|DiagCross|UTriangle|LTriangle` (the specification is case insensitive, e.g. `shape=diagcross`)
- `labels=`*columnName* (or `label=`)
- `regression=linear|logarithmic|no`
- `style=dots|connected|splines|bars|triangles`
- `xerr=`*columnName* # X axis confidence interval
- `yerr=`*columnName* # Y axis confidence interval
- `tooltip=col1,col2,...` Comma separated list of column names for balloon popup when user move mouse over the curve dot.

Histogram plot arguments

The syntax of the *columnName* can refer to one or multiple columns as follows: *columnName*|{*columnName*[,*columnName*...]}

A distinct *columnName* can be further split by another *columnName* as such: *columnName*[*:columnName*]

- ♦ x=*columnName* or y=*columnName* # must be present
- ♦ pinwheel=*color*[*/color*...] (e.g. make plot t "x={A,B,C,D};pinwheel=red/blue"
- ♦ step=*bin size* (e.g. make plot t "x=A;step=20"
- ♦ binWidth=*relativeBinWidth[0.,1.]* # (saved on exit)

Examples:

```
group table t Random(1. 10. 40) Random(1. 10. 40) Iarray(20,0)//Iarray(10,1)//Iarray(10
make plot t "x={A,B}"
make plot t "x=A:C"
```

General Properties (all are optional)

- ♦ title=*plot_title*
- ♦ xTitle=*X-axis label*
- ♦ yTitle=*Y-axis label*
- ♦ grid=yes|no # grid lines
- ♦ axes=no|yes # additional X=0 and Y=0 axes
- ♦ xStep=*x* # x tick marks
- ♦ yStep=*y* # y tick marks
- ♦ xRange=*fr:to* # shows only the plot in the specified range
- ♦ yRange=*fr:to* # shows only the plot in the specified range
- ♦ scale=*scale* # is updated upon interactive rescaling

Properties of the geometrical elements (rectangle,ellipse and polygon)

- ♦ color=*color*. The *color* is a name or a hexadecimal rgb string, e.g. color=red or color=#ff0000
- ♦ rotate=*angle_degrees* (e.g. rotate=45 means 45 deg. counter-clockwise)
- ♦ fillStyle=BDiagPattern|SolidPattern|HorPattern|VerPattern|CrossPat
- ♦ label=*My label* # each semicolon or backslash inside the label needs a preceding backslash, i.e. **\;**, \\ .
- ♦ labelPos=center|left|right|top|bottom

Example

```
group table t {300. 200. 500.} "Volume" {390. 230. 630.} "Area" {5 3 1} "i"
x = "x=Volume;y=Area;color=i;size=8;;title=Volumes and areas;;"
y = "element=rectangle;x1=150;y1=200;x2=550;y2=550;color=blue;fillStyle=BDiagPattern;la
make plot t x+y
```

Option *output* writes the plot as an image into file name provided. File extension defines the image type. (Most popular extensions: png,jpg,pdf,eps) With output option you may additionally specify the size of the output image.

Example:

```
 add column t Random(100,0.,1.) Random(100,0.,1.) Random(100,0.,1.)
 make plot t "x=A;y=B;color=C;size=6;style=dots;;" output="aaa.png" size = {500,500}
 unix display aaa.png   # display the plot using Linux 'display' program
```

Visualizing a matrix

Element matrix=*matrixName* with additional arguments:

- ♦ depth=
- ♦ legend=
- ♦ pos=
- ♦ step=
- ♦ rainbow=

General properties are also applicable after two semi-colons: (xTitle , yTitle , title)
Example:

```
read pdb "1crn"
m = Matrix(a_/A a_/A ) # residue contact matrix
add header t m name='m'
make plot t "matrix=m;rainbow=white/yellow/green;depth=-1;legend=m;;xTitle=I;yTitle=J;t
```

See also:

- ♦ `delete plot` # to delete plots

make reaction : applying reaction to the products

`make reaction` *reaction_R1R2 chem_R1 chem_R2* .. [`name=`*s_table* | `output=`*s_file*] [`filter=`*s_expression*] [`all` | `only`] [`keep`]

Takes one chemical reaction (the first element of a `reaction` array) and applies it to the reactant arrays. All reactants suitable for this reaction will be combined to generate a chemical `table` . The **reaction** drawing need to refer to replacement groups as `R1` , `R2` , for example:

```
Parray("[R1]C(=O)O.N[R2]>>[R1]C(=O)N[R2]" )
```

Note that here the dot **.** separates the reactants and the >> indicate the reaction. Example:

```
m1 = Parray( { "C(C(=O)O)(=C(C=C1)N)C=C1SC#N" "C(=CC(=C(O)C1)C=CC=1)(C(=S)N)C(=O)O"\
   "N(=C(S1)N)C(C1=CC(C=C1)=CC=C1C(=O)O)=O" "C(C(=O)O)(C(=CC1)N)=CC=1N=C=S" } )
m2 = Parray( { "C(C(=O)O)(=C(C=C1)N)C=C1SC#N" "C(=CC(=C(O)C1)C=CC=1)(C(=S)N)C(=O)O"\
   "N(=C(S1)N)C(C1=CC(C=C1)=CC=C1C(=O)O)=O" "C(C(=O)O)(C(=CC1)N)=CC=1N=C=S" } )
make reaction Parray("[R1]C(=O)O.N[R2]>>[R1]C(=O)N[R2]" )  m1 m2
```

`keep` options preserves `SMARTS` search attributes in the result

`all` and `only` options allows one to handle multiple functional group mapping. The possibilities are:

- ♦ default mode : skip compounds with multiple functional group matches
- ♦ `only` option : take a first mapping
- ♦ `all` : enumerate all possible mappings

You can apply a filter expression to the output of the reaction. The following functions can be used in the filter:

MolWeight,Nof_Molecules,Nof_Chirals,Nof_RotB,Nof_HBA,Nof_HBD,Nof_Atoms,Nof_Frags,DrugLikenes

Example:

```
make reaction Parray("[R1]C(=O)O.N[R2]>>[R1]C(=O)N[R2]") m1 m2 filter="MolWeight<400 &
```

You can also apply condidtion to the matched functional groups. *R1,R2,...* Example:

```
react = Parray("[R1]C(C(=O)[R2])=O>>C1C(=C(C=CC=1)[R2])[R1]")
rct1 = Chemical( {"CC(C(C)=O)=O", "CC(C(c1ccccc1)=O)=O"} )
make reaction react rct1 filter = "R1==R2"
make reaction react rct1 filter = "R1!=R2"
```

Another example (provided as an example file):

```
read binary s_icmhome + "example_reaction1.icb"
make reaction r_han_pyr.rxn[1] name=Name( "T_react", unique ) reactant1.mol,reactant2.m
```

The output. The output is generates as a table if the total number of products is less than 40,000. For larger sets the command will automatically switch to the file mode.

- ♦ Option `name` = *s_table* allows one to change the default name of the output table.
- ♦ Option `output` = *s_file* forces the file output and suppressed the table creation.

The output chemical table has the product column as well as a column for each of the R-groups.

Dynamic filtering of the output by applying a `filter` expression. The `filter=` *s_expression* option allows one to apply a filter during the library generation. The filter expression is a double-quoted string with the following structure: "*Function1 relation value* **&** or **|** *Function2 relation value* **&** or **|** .. "

Example:

```
filter = "MolLogP<5. & Nof_Frags('C(O)=O')<1"
```

The list of functions is expanding. The current list of the functions is the following:

Function Name	Description	Example

MolWeight		MolWeight < 650
Nof_Molecules	the number of individual molecules,including ions and salts	Nof_Molecules==1
Nof_Chirals	the total number of racemic and chiral centers	Nof_Chirals==0
Nof_RotB	rotatable bonds	
Nof_HBA	hydrogen bonding acceptors	
Nof_HBD	hydrogen bonding donors	
Nof_Atoms	the total number of non-hydrogen atoms	
Nof_Frags (*s_smart*)	counts the number of fragments	Nof_Frags('[S,P](=O)=O')==1
DrugLikeness	a number around 0	DrugLikeness > 0
MolLogP	log P prediction	
Volume	3D molecule volume prediction	
MolPSA	polar surface area	
MoldHf	heats of formation	
MolLogS	solubility	

see `Predict` for a detailed description of some of the functions.

Splitting the library by the scaffold into the replacement group arrays.A library can be also reduced back to the scaffold and replacement groups using the `split group` *scaffold library* command. E.g. `split group scaffld.mol combilib.mol`

See also: `enumerate library` , `Split chemical` , `Replace chemical` .

make sequences from alignment

`make sequence` *ali_range*
take a vertical block from an alignment and convert it into a separate alignment with independent truncated sequences.

```
read alignment s_icmhome+"sh3"
make sequence sh3[20:50]
```

See also:

- ♦ `delete sequence compress` - removes sequences not used in alignments
- ♦ `Align`

make sequence: extract from pdb or icm structure

`make sequence` *ms* [`name=`{*s_name* | *S_names*}] [`resolution`]
creates sequences (or, more strictly speaking, ICM-shell objects of the 'sequence' type) of residues composing selected molecules *ms_* . One-letter equivalents of full residue names are specified in the `icm.res` library. Option `resolution` adds the X-ray resolution value multiplied by 10 to the name (e.g. 2ins_a25 for resolution of 2.5A) or `'No'` for NMR and theoretical structures. The `group sequence` command will automatically prefer a sequence from structure with better `resolution.` The resolution is not appended if option `name=` is specified.
The `make sequence` command also extracts both the secondary structure and the `site` information.
See also: `read pdb sequence`
Examples:

```
 read pdb "2ins"
 make sequence a_2ins.a,b        # two seqs 2ins_a and 2ins_b created
 make sequence a_2ins.a,b resolution # resolution*10 added to the name
 make sequence a_1.1 name="aa" # sequence named: aa
 make sequence a_2ins.a,b name={"aa","bb"} # seqs named: aa and bb
```

make tree

`make tree` *table* [*tree_type*] [*tree_name*] [*options*]

this command builds a hierarchical data tree of the table rows and stores in the table header. as a *tree*`.cluster` parray. A simple example is given below:

```
read table "t.tab" name="t"
make tree t
```

Defining the tree construction type.

The main modes are the following:

- ♦ a complete tree requiring N^2 comparisons. It is preferable for tables smaller than a thousand records. For larger tables both performance and memory requirements become prohibitive. This mode will be used by default for up to 2000 rows. To force this mode use keyword `full`, i, or the method name.
- ♦ a *K-means* clustering requiring ***N*K*** comparisons. To activate this mode the number of clusters *i_Kclusters* needs to be specified. This method is both much faster and does not require a pre-existing ***NxN*** distance matrix.

For a small number of rows (*N* under two thousands) ICM performs data clustering based on *N* by *N* comparison between the table rows. This method can be enforced for larger tables with the `full` option (e.g. `make tree t full`) or using the `distance` = *s_distMatrixName* . In the latter case the matrix needs to be appended to the table header:

```
read table "t.tab" name="t"       # N rows
make tree t full                  # all distances between rows are calculated
# or
read matrix "dist.mat" name="dm"  # N x N matrix
group table t append header dm
make tree t distance = "dm"       # uses external distance matrix for clustering
```

The **type** (*tree_type*) of the cluster tree construction algorithm can also be explicitly defined. This type defines how two distances from two neighboring branches are replaced with one distance. By default the `"upgma"` method is used.

- ♦ `"upgma"` (the *default) unweighted pair group method using averages*, i.e. each record has the same weight, and therefore, the distance to each branch is weighted by the branch size: ***d12=(N1*d1+N2*d2)/(N1+N2)*** .
- ♦ `"single"` or `"min"` single linkage, if two branches with d1 and d2 are merged, the distance is replaced with the minimal distance: ***d12=Min(d1,d2)*** .
- ♦ `"complete"` or `"max"` , e.g. ***d12=Max(d1,d2)*** .
- ♦ `"wpgma"` : average linkage tree, each branch has equal weight regardless of the number of members: ***d12=0.5*(d1+d2)*** .

If the number of rows *N* is large, ICM performs the **K-means** clustering by default. The K-means method can also be enforced if the number of clusters is provided as an explicit integer argument, e.g.

```
make tree t 100 # enforces K-means with 100 seed clusters.
```

This method avoids a full comparison by creating *K* seeds and comparing all other records with the seeds. If neither `full` argument, nor the *type* string, nor the number of seeds for the *K-means* clustering is specified, the method of construction will be determined automatically on the basis of the number of rows.

Defining the distance measure.

The distances between rows can be (1) provided externally in the table header (option `distance=s_matrixName`, see below) (2) calculated dynamically as Tanimoto distances between chemical fingerprints for chemical tables; (3) calculated dynamically as weighted cartesian distances (options `column=` *S_listOfColumns* and `heavy=` R_listOfColumnWeights). The tree can further be used to assign a cluster number to each table row at a certain distance threshold (the `separator=` and the `split=` options).

For common tables, by default all numerical tables are used with the same weight. For chemical tables, by default only the chemical array is used for distance calculations. However, option `all` will enforce concurrent use of the chemical distances and all numerical columns. Options `column=` and `heavy=` give a specific selection of columns and their respective weight.

- ♦ `all` add all numerical columns to the chemical information in distance calculation
- ♦ `column=` *S_listOfColumns* numerical column used for clustering
- ♦ `distance=` *s_distMatrixNameFromTableHeader* . The matching distance matrix must be already attached to the table with the `group table` *table* `header` *matrix s_distMatrixFromTableHeaderName* . This only belongs to the `full` tree mode.
- ♦ `exact` suppressed auto-normalization fo columns in calculating cartesian distances during clustering
- ♦ `heavy=` *R_distColumnWeights* numerical column weights corresponding to the `column=` column names
- ♦ `label=` *s_labelFormat* (e.g. `label= "%COMPOUND_NAME"`)
- ♦ `matrix` add the newly formed distance matrix to the header of the table (the default name is `"dimt")`.This only belongs to the `full` tree mode.
- ♦ `name=` *s_treeName* defines the name of the tab associated with this tree (the same table can have several associated trees)
- ♦ `select=` *I_rowNumbers* (e.g. `select=Count(10,30)` this option allows one to create a tree from a subset of rows, e.g. from 10 to 30.
- ♦ `separator=` *r_threshold_Dist* . The minimal distance value at which elements or clusters are considered to belong to the same group.
- ♦ `sort=` *s_orderColName* additional ordering of the branches by a table column with preservation of the clusters (`sort="A"`). It is possible because *left* and *right* is normally undefined. If the *s_orderColName* is specified, the *left* and *right* will be determined by this column values.
- ♦ `split=` *s_colname* : re-calculate the cluster numbers at a different threshold value (a.k.a. `separator)` . *s_colname* is the name of a column in which the new cluster number is stored, the "splitting" is done according to the `separator` value. This split level can be reset with the `split tree` command, and the column name can be returned with the `Name(` *table*`.cluster` *i_cluster* `split`) function.

The output.

- ♦ The tree is added/appended to the *table*`.cluster` parray.
- ♦ `i_out` returns the position index in *table*`.cluster` .
- ♦ a new column is added to the table with the order number of records in the tree. This column can be used to sort the table rows in the tree order. The name of this column can be found with the `Name(` *table*`.cluster` *i_cluster* `index`) function.
- ♦ another column containing the cluster number at the `split` level is added if the split argument is used (see above).

Example:

```
read table mol s_icmhome+"/moledit/Dictionary.sdf"
make tree Dictionary
```

See also:

- ♦ `delete variable` *tree_Parray*
- ♦ `Name` (*tree_Parray* ..)
- ♦ `split`

`make tree` *ali_name* [*s_epsFile*]
reconstruct the evolutionary tree from the specified sequence alignment using the neighbor-joining method (`Saitou and Nei, 1987`). Create a PostScript image of this tree which will be saved in the *ali_name*`.eps` file. See also: the `align` command.
Examples:

```
 read alignment msf s_icmhome+"azurins" # read alignment
 make tree azurins              # draw evolutionary tree
```

`make tree` *M_squareDistanceMatrix*[`1`:`n`,`1`:`n`] [*S_objectNames*[`1`:`n`]] [*s_epsFile*]
reconstruct the evolutionary tree for arbitrary objects from the **matrix of pairwise distances**. The names of individual objects may be provided in a string array for a nicer PostScript picture. This command is cool.
See also:

- ♦ `Disgeo` function.

Examples:

```
 read matrix s_icmhome+"dist"           # read a distance matrix [n,n]
 make tree dist
 unix gs dist.eps
```

make tree object

make tree object *M_dist_nxn* [*S_names_n*] [name=*s_objName*] [angle=(90:180)] [torsion=(0:180)] [simple] [size=*r_distScale*]

makes a 2D (option simple) or 3D tree and sets an integer field named 'index' with the index value to each atom (see set field and Field(*as s_fieldName*))

This example shows how to start from a table and make an active tree (nodes respond to doubleClick) :

```
read table mol s_icmhome + "FUNCGROUPS.sdf" name="t"
make tree object Distance( t.mol ) name = "tree"  # sets "index" field for leaf atoms
# set or re-set labels
set label atom a_//c* Sarray(t.NAME_ [ Field( a_//c* "index" ) ])
# set ball radius and colors
set atom ball a_//c* t.clogP [ Field( a_//c* "index" ) ]
color ball a_//c*  t.tPSA [ Field( a_//c* "index" ) ]
# set double click property (e.g: to select an corresponding table row)
set field a_//c* name="doubleClick" "find table t select index=Field( $1 'index')"
```

make unique: reorder atoms in a unique order.

Example:

```
read mol s_icmhome+"ex_mol"
make unique
#
build hydrogen
set type mmff
convert
Smiles(a_)  # unique smiles string
```

minimize

minimize [vs] [*i_mncalls*] [*s_terms*] [selftether=*as*] [*as_1* [*as_2*]] \n\

minimize cartesian|mmff [type] [charge] .. \n\

minimize stack [selftether] .. \n\

minimize tether [disulfide] .. \n\

minimize locally the sum of currently active, or specified, terms of the energy/penalty function with respect to variables specified by *vs_*, or all the free variables, if variable selection is skipped.
Optional arguments:

stack : If option stack is specified, the procedure extracts each stack conformation, minimizes it and replace the stack conformation with the optimized ones. The stack can be generated with the montecarlo procedure, manually created with the store conf command, or read from a stack file. This command allows one to refine your set of alternative conformations all at once.
i_mncalls : defines the maximal number of iterations. The minimization procedure can terminate earlier if the gradient becomes lower than the tolGrad parameter. If *i_mncalls* is not provided, the default parameter mncalls defines the maximal number of function evaluations during the minimization.
vs_ : variable selection If selection of variables *vs_* selection is specified, the object will be refixed but the initial fixation will be restored after minimization.
s_termString : redefines the set of terms used in the minimization dynamically (e.g. minimize "tz"). You may check the active terms with the show terms command, or change them before the minimization with the set terms ".." command. By the way, the active terms can be shown as a part of your command line prompt if you add the %e specification to

`s_icmPrompt` variable (like `s_icmPrompt="icm/%o/%e> "`).
`selftether=` *as_* : if term "ts" (tether to self) is active, you can select a subset of atoms to be tethered

atom pair filter: By default all the atoms and all the atom pairs within distance thresholds `vwCutoff` and `hbCutoff` are involved in the calculation. However, two explicit atom selections [*as_1* [*as_2*]] may impose a mask on atom pairs involved in the calculation of the pairwise energy or penalty terms. The default for the skipped *as_1* is all the atoms. If only the *as_1* is specified, the *as_2* is assumed to be all atoms. Using atom selections is dangerous and is not recommended since there are many combinations which do not make sense and give unpredictable results.
the algorithm: the `minimizeMethod` preference. The type of algorithm (conjugate gradient, quasi Newton, or automatic switching between the two) is defined by the `minimizeMethod` preference. The progress bar will show you the progress of the procedure. If minimizeMethod="auto", the progress bar of the minimization procedure will show the 'C' character in a row of dots and colons when the quasi-Newton method switches to the conjugate gradient method.
Dots show progression of the minimization procedure, while colons mark recalculations of neighbor lists. The lists are updated if at least one of the atoms deviates from its previous position by more than 1.5 A. Both basic methods use the analytical derivatives of the terms with respect to free internal variables.
the exit criteria, and interaction lists. The procedure is terminated upon any of these conditions become true:

- `mncalls` : the maximal number of function evaluations (`mncalls) is reached.
- `tolGrad` : if the gradient falls below the `tolGrad` parameter.
- `tolFunc` : if all of 6 consecutive steps do not improve the function beyond the `tolFunc` parameter.
- `minNumGrad` condition : this is a condition when the **num**erical gradient evaluated from the energy function values differs too much from analytical gradient. It is usually the case when the minimum is essentially reached, but the atoms bumped into each other and the slope is steep (R^12). Naturally, if the function has a strong non-harmonic behavior (e.g. a packed protein) and the **num**erical gradient does not match the analytical one. This is not necessarily bad, just means that you reached a packed state. If you rerun the minimizer, it may go in a different direction and may improve the function a little more.

Suppressing updates of the interaction lists upon atom movements during the minimization. Sometimes during the course of minimization the interaction lists are recalculated. When some atoms fall out or in the `vwCutoff` sphere , the value of the gradient and even the energy function can jump. For that reason do not be surprised that the exit gradient differs from the one reported in the previous step output. To influence the lists you have two main mechanisms:

1. suppress list updates all together with `l_updateLists` = `no` . In this case, if you need to recompute the lists, use the `delete list` command.
2. make the updates less frequence by increasing the `listUpdateThreshold` parameter.

Example:

```
read object s_icmhome+"crn"
l_updateLists= no
minimize
show energy # still uses the same lists
delete list
show energy # makes new lists

l_updateLists= yes
listUpdateThreshold=2.
minimize
```

the output `l_showMinSteps` flag and `i_out` : The actual number of function evaluations during minimization is saved in the `i_out` variable. The `l_showMinSteps` flag allows one to see every iteration of the minimization procedure. To speed up the procedure you may switch off the `l_minRedraw` flag to suppress redrawing of the molecule for each new conformation.

The minimizer also returns a `rarray` `R_out` with the following values:

- `R_out[1]` the return code as follows:
  - ◊ `0` successful completion,
  - ◊ `1` `mncalls` expired

- ◊ 2 `tolGrad` is reached
- ◊ 3 `tolFunc` is reached
- ◊ 4 tolXdiffOK
- ◊ 5 minBadGrad
- ◊ 6 minimization was interrupted
- ◊ 7 the minimizer is lost in high energy values
- ◊ 8 whatever you do, it goes uphill
- ◊ 9 unknown failure

♦ `R_out[2]` the number of function calls spent on the minimization run
♦ `R_out[3]` the norm of the gradient (rmsd) upon completion

Examples:

```
build IcmSequence("HHAS;TW")  # create object from "def.se" sequence file
minimize v_//xi*              # do not touch the backbone torsions
minimize                      # use all variables
minimize 500                  # run longer until number of calls is 500
```

minimize cartesian: full conformational optimization

`minimize cartesian` [`stack`] [`type`] [`charge`] [*i_mncalls*] [*s_termString*] [`selftether=`*as_for_ts_term*]
minimize the `mmff` energy for a fully flexible molecule in the space of atomic cartesian coordinates. Before running this command please make sure that the atomic types and charges are set and the `mmff` libraries are loaded.
The *i_mncalls* and *s_termString* have the same meaning as in the previous command. Options:

♦ `stack` : if option `stack` is specified, the procedure extracts each `stack` conformation, minimizes it and stores back to the stack.
♦ `type` : if option `type` is specified the `set type mmff` command is executed and mmff atoms are assigned.
♦ `charge` : if option `charge` is specified the `set charge mmff` command is executed and mmff partial charges are assigned
♦ *i_mncalls* : redefines the maximal number of minimization iterations (`mncalls`)
♦ *s_termString* : allows one to dynamically redefine the default energy terms.
♦ `selftether=` *as_for_ts_term* : if `term "ts"` (tether to self) is active, you can select a subset of atoms to be tethered

Example:

```
build string "se nter his cooh"
display
set term "ts" # tether to the initial set of coordinates
minimize cartesian type charge selftether=a_//ca,c,n
```

The `drop` and `tolGrad` minimization parameters will still apply.

minimize loop after build model

`minimize loop` *i_loopNumber*
to use this command you must run the `build model` command first. The `build model` command may not be able to find a perfectly matching loop. Two sorts of problems may appear: the imperfections of the loop attachments and the clashes of the loop to the body of the model. The `minimize loop` command optimizes the covalent geometry at the junctions and the clashes through an interactive procedure which maintains the loop closure.
The energy function used by the command is not as detailed as the full atom energy. It is advisable to perform a regularization (e.g. `regul a_`) and full atom refinement.
To save all the graphical frames during this minimization set the `autoSavePeriod` variable to the special value of `99` . In this case png image files named `f_x_y.png` , where `x` is the loop number and `y` is the frame number, will be saved in the current working directory.

minimize stack: minimize each stack conformation

`minimize stack` [*s_terms*] [`mncalls`]
execute these steps:

1. `load` each `stack` conformation
2. locally `minimize` it
3. `store` each conformation back to the stack

As a result, both the geometries and the energies are updated with the optimized ones. Example:

```
read stack "a"
minimize stack 400
```

One can achieve the same result with a shell script like this:

```
read stack "a"
for i=1,Nof(conf)
  load conf i
  minimize 400
  store conf i
endfor
```

minimize tether: threading a model with idealized geometry through a pdb-structure

`minimize tether` [*vs*]
regularization procedure. It creates a conformation (i.e. determines free variables) that minimizes distances between atoms and their tethering points. If initial model was built from standard amino-acids with idealized covalent geometry , this procedure will create a model with standard bonds and angles which fits the best to the target set of atom coordinates. The `tethers` may be imposed by the `set tether` command. An integer variable `minTetherWindow` defines the maximal number of preceding torsions which are locally minimized to best-fit the pdb-model. Optional variable selection *vs_* allows one to perform fitting only for the selected fragment of the model. This may be convenient if you want to re-fit only a local fragment. Variable `r_out` contains the RMS deviation between the template and the model.

Assigning ring conformation from a template

To assign ring conformation from a template one can use the `minimize tether` command. The chemical equivalences can be found and tethers imposed with the `find molecule sstructure all tether` command. The following example illustrates the principle.

```
read object "template.ob" name="template"  # contains ring template
build smiles "C1CCCCC1"  # some ring
find molecule sstructure all tether a_template.  a_target.  # make sure tethers exist
set object a_target.
unfix V_//r*,f*
minimize tether
minimize cartesian "mmff,ts" selftether=a_//!h*
display a_template,target. center
```

menu

a tool for making clickable strings in the graphics window.
`menu` [*i_string1 i_string2 ...*]
this command declares the listed string labels as active and returns the chosen string number in `i_out` . If no arguments are specified, only the last string will be "clickable". See also _demo_main file.
Examples:

```
 while(yes)
 display string "Menu"
 display string "Fish"    -0.7, 0.6   yellow # 2
 display string "Pork"    -0.7, 0.5   yellow # 3
 display string "Pasta"   -0.7, 0.4   yellow # 4
 display string "Quit"    -0.7, 0.3   yellow # 5
 menu 2 3 4 5
 choice=i_out
 delete label
 if    (choice == 2) then
   display "Good choice.\n Our fish is the best.\nClick here"
   menu
   delete label
 elseif(choice == 3) then
   display "Good choice.\n Our pork is the best.\nClick here"
   menu
   delete label
 elseif(choice == 4) then
   display "Good choice.\n Our pasta is the best.\nClick here"
   menu
   delete label
 elseif(choice == 5) then
   quit
 endif
 endwhile
```

modify

modify chemical structure of a molecule by replacing one part with a specified group or "residue" from `icm.res` or user residue library. Prerequisites:

- ♦ modify works only for ICM objects. `convert` your object to ICM type if necessary
- ♦ modify deletes the atoms which need to be replaced, so you do not need to delete them explicitly

modify atom with a library group
`modify` *as_exitAtom s_graft_branch*
replace the branch starting from the specified atom by another library substituent. Suitable for standard biochemical modifications, such as glycosylation, phosphorylation, etc. (Note that to myristoylate N-terminus you need to use "myr" as N-terminal residue, i.e. `build string "se myr ala ala coo-"`).
Examples:

```
 LIBRARY.res = LIBRARY.res // "usr"        # Use usr.res in s_icmhome in addition to ic
 read library residue                      # Re-Read the library with additional residu
 read object s_icmhome+"crn"
 display a_/8:13
 color red a_/11                           # serine
                                           # O-glycosylation ("bnag", "bgal", "bglc", "
                                           # hint type Table(residue) to see available
 modify  a_crn.m/11/og "bnag"              #  beta-D-N-acetylglucosaminide
 # Or
 build string "se ser thr tyr asp lys his"
 modify a_/ser/og  "po4"                   # Phosphorylation
 modify a_/thr/og1 "po4"
 modify a_/tyr/oh  "po4"
 modify a_/asp/od2 "po4"
 modify a_/lys/hz2 "po4"
 modify a_/his/hd1 "po4"
```

See also: `LIBRARY.res`

modify: single or multiple residue mutations `modify` *rs s_NewResidueName*

replace selected residue(s) `rs_` by another residue *s_NewResidueName.* The backbone conformation is not changed, unless the new residue is "pro" and the phi angle is outside [-90.,-30.] range.
You can replace amino acids (the usual list of three letter codes), as well as nucleotides: `"ra" "rg" "rc" "ru"` for RNA and `"da" "dg" "dc" "dt"` for DNA.

Examples:

```
# Peptides and proteins
#
 read object s_icmhome+"crn"
 modify a_/15,18 "his"          # substitute residue 15 and 18 with histidines
 modify a_/thr "val"            # substitute all alanines with valines
#
# DNA or RNA
```

```
#
 read pdb "4tna"
 convert
 modify a_/66 "dg"        # substitute nucleotide 66 by Uracyl
```

Modifying the 1st residue in a polypeptide

The first residues has an unusual N-terminus therefore there is a special trick to mutate it to another residue. Essentially the grafting principle (see below) needs to be used. Example in which we are modifying 1st residue of the first object:

```
build string "HWT" name="x"; align number a_/* 0 # our main object
#
build string "ACDEFGHIKLMNPQRSTVWY"  name="allres"  # the source of replacement groups
modify a_1./his/cb  a_allres./arg/cb   # modifies the side chain only
```

modify by grafting parts of objects `modify` *as_atom1 as_atom2*
replace a fragment of the molecular tree in an ICM-object starting from a specified single atom *as_atom1* (e.g. a_/15/cg) by a subtree starting from another single atom *as_atom2.* This subtree is simply copied and not altered in any way. It is recommended to perform molecular building operation interactively and with your molecule displayed in the graphics window. Type `modify` and `Ctrl-click` the atom starting the branch to be replaced and then the atom starting the branch to be grafted. It does not matter where you take the modification group from. It may be the same molecule, a group in another object, etc. You may want to load a residue containing the group of interest directly from the `icm.res` residue library by doing.
Examples:

```
 show residue types     # find out what residues are available
 build string "se myr"  # create a new object with myristoyl group.
```

After the modification you can remove objects (such as `"myr"` in the above example) used for construction. Be careful if modifying atoms within ring systems; the results may not always be obvious unless you know how the ICM-tree is constructed (you'll be kindly warned anyway). However, the whole ring can be modified or grafted without any difficulty.
Examples:

```
 build string IcmSequence("MIPEAY")  # build a molecule
 display                             # display it to click two atoms and watch

 modify a_/1/ce a_/1/ha              # replace methyl group of Met-1 by a hydrogen
 modify a_/2/hd13 a_/2/cg2           # methylate hd13 hydrogen of Ile
 modify a_/3/hg1 a_/6/oh             # turn proline into hydroxyproline
```

See also: `chemical modification of chemical arrays`

Circular permutation of x,y,z coordinates and cell parameters

`modify rotate` [*os_nonICM*] [*i_n_perm*(`1`)] [`only`]

performs one or two (if *i_n_perm* is 2) circular permutation of Cartesian coordinates of atoms and unit cell parameters a,b,c and alpha,beta,gamma parameters of the unit cell.
Option `only` suppresses the cell parameter permutation.and only does x,y,z. This command has been developed to fix problems with incorrect (nonstandard) definitions for C121 space group. Normally the second angle (beta) is supposed to be non-90 (is the case for 6000 PDBs with C121 groups), while in the following list

```
 7acn 8acn 1aco 1ami 1amj 1b0j 1b0k 1fgh 1gra 1grb 1gre 4gr1 1grf 4grt 1grg 5grt 1grh 2
 3grt 1lh1 1lh2 1lh3 1lh5 1lh6 1lh7 2lh1 2lh2 2lh3 2lh5 2lh6 2lh7 1nis 1nit
```

the **third** angle (gamma) is non-90.

Example:

```
read pdb "1gra"
findSymNeighbors a_ 7. no 2 yes yes    # there is a problem with symm generated objects
```

```
#
delete all
read pdb "1gra"
modify rotate
findSymNeighbors a_ 7. no 2 yes yes    # problem solved
```

Chemical modifications.

find and replace a chemical pattern, normalize/standardize a chemical, delete salts The following types of chemical modifications can be performed on an array of chemicals:

- `modify` *`chem_array s_from s_to`*
- `modify` *`chem_array`* `delete salts`
- `modify` *`chem_array`* `auto`
- `modify` *`chem_array s_frag1 s_frag2`*

Chemical Find and Replace

`modify` *chemarray s_find_smart s_replacement_smart* [`exact`] [`index=` *I_indices*]

find a chemical pattern and performs a global replace to *s_replacement_smart* for all chemicals in an array. The replacement can be done only if both the find and replace patterns contain the same marks `R1,R2`.. A connecting atom in the pattern can be either an undefined atom, e.g. `[R1]` or a defined atom, e.g. `[C;R1]` These marks are used to find corresponding atoms in the replacement pattern.

Hints for the Chemical Editor: use the following shortkeys:

- to mark atoms as `R1` point your cursor at the connection atom and press 1
- to retain the identity of the attachment atom, (e.g. `[C;R1]`) use `Ctrl-1` ..
- to undo the `R1` mark, label it as N,C,or O; press `Ctrl-0` for `[C;R1]` .

E.g:

```
read table mol s_icmhome+"/moledit/Dictionary.sdf"
modify Dictionary.mol "[R1]CC(=O)O" "[R1]CC(=O)OC"
```

Option `exact` modifies atoms in place and requires that the number of atoms is preserved.

Charge or uncharge functional groups in compounds

`modify` *chemarray s_find_smart s_replacement_smart* [`exact`] [`index=` *I_indices*]

searches for a chemical group and replaces it by a group with a different charge. Add `index=` *I_idx* if you want to apply the operation to a selection.

Carboxylic_Acids

`modify` *chem* `"CC(=O)[O;D1]" "CC(=O)[O-]" exact # to charge`

`modify` *chem* `"CC(=O)[O-;D1]" "CC(=O)[O]" exact # to uncharge`

Primary_Aliphatic_Amines

`modify` *chem* `"[C;^3][N;D1]" "C[N+]" exact # to charge`

`modify` *chem* `"[C;^3][N+;D1]" "C[N]" exact # to uncharge`

Secondary_Aliphatic_Amines

`modify` *chem* `"[C;^3][N;D2][C;^3]" "C[N+]C" exact # to charge`

`modify` *chem* `"[C;^3][N+;D2][C;^3]" "C[N]C" exact # to uncharge`

Tertiary_Aliphatic_Amines

modify *chem* "[C;^3][N;D3]([C;^3])[C;^3]" "C[N+](C)C" exact # to charge

modify *chem* "[C;^3][N+;D3]([C;^3])[C;^3]" "C[N](C)C" exact # to uncharge

Amidinium/Guanidinium

modify *chem* "[N;D1]C=[N;D1]" "NC=[N+]" exact # to charge

modify *chem* "[N;D1]C=[N+;D1]" "NC=[N]" exact # to uncharge

#

Chemical Find and Replace

modify *chemarray* {delete|split} salt[= *chemarrayWithSalts* [add]] [simple]

Removes salts using dictionary file $ICMHOME/SALTDICT.sdf.

Example:

```
add column t Chemical("CCC.CCC.O.NN" )
modify chemical t.mol delete salt  # uses default salt dictionary
```

New or additional salt patterns can be provided with `salt=`.

Example:

```
add column t Chemical("CCC.CCC.O.NN" )
modify chemical t.mol delete salt=Chemical({"O","NN"})  # treats both 'NN' and 'O' as s
```

with `add` option both default and used provided dictionary will be used.

`split` option created two new columns in the original table 'salt_smiles' and 'salt_names' with dot separated smiles and names for removed salt.

`simple` option toggles mode which retains only the largest molecule and deletes all smaller molecules. Example:

```
c = Parray( {"CC=O.O"})  # has an extraowater, O
modify c delete salt
 Info> 1 replacements done
show c
 CC=O
```

Standard representation of chemical groups

modify *chem_array* auto

applies a set of chemical normalization rules described in the `CHEMNORMRULES.tab` in the $ICMHOME directory . Feel free to add rules to this table or replace it with your own table. Currently it has the following rules:

```
"*-[N+](=O)[O-]"          "*-N(=O)=O"          "Nitro"
"*-N([OH])[OH]"           "*-N(=O)=O"          "Nitro"
"*-N(=O)[OH]"             "*-N(=O)=O"          "Nitro"    # Sulfonyl
"S(=O)([OH])(-*)(-*)"     "S(=O)(=O)(-*)(-*)"  "Sulfonyl" # Azide
"*-[N-]-[N+]#N"           "*-N=[N+]=[N-]"      "Azide"    # Diazo
"[C-]-[N+]#N"             "C=[N+]=[N-]"        "Diazo"
```

The asterisk marks "any atom". Note that if the pattern does not contain connection labels `[R1]`, or `[C;R1]` , the atoms will not be considered as terminanted. A more general syntax is described in the `chemical find and replace` command.

See also:

- file `CHEMNORMRULES.tab` in the ICM home directory

Update database table from ICM table

`modify molcart` *T* [`all`] [`column=`*S_columns*] [`table=`*s_table_name*] [*connection_options*]

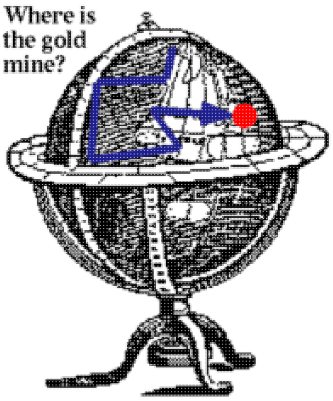
Modifies entries in a database table based on changes made in an ICM table. The `load molcart`, `find molcart` and `query molcart` create ICM tables containing subsets of database tables. In most cases the database table has an integer primary key (ID) column, with unique values for each row. The primary keys may be used to mark certain rows in the ICM table to request update or deletion of the corresponding entries in the database. ICM GUI provides tools to mark rows.

The `all` option tells the command to treat all rows as marked for update. Only a subset of the ICM table columns may be updated in the database by specifying *S_columns.*

The connection may be specified using *`connection_options`* . Database table is specified Otherwise connection and table information may be obtained from the input table header as in the `load molcart` command.

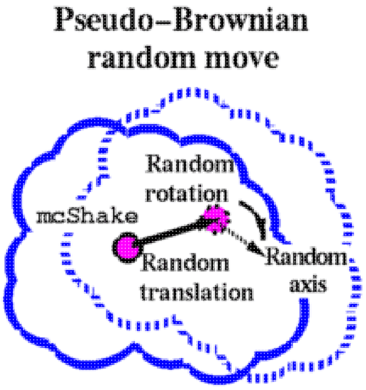
See also: `molcart`.

montecarlo



a generic command to sample conformational space of a molecule with the ICM global optimization procedure.

`montecarlo` [`OPTIONS`] [*vs_MC* [*vs_minimize*]] [`local` *rs_loop*]
runs Monte Carlo simulation for specified variables *vs_MC*, with local minimization with respect to the *vs_minimize* variables following after each random move.



Each iteration of the procedure consists of

1. a random move of one of 4 types;
   - ◊ change one internal variable by a random value (e.g., montecarlo v_//x*)
   - ◊ change a group of angles described as a `vrestraint` according to its probability distribution (e.g. set vrestraint a_/* ; montecarlo v_//*)
   - ◊ change the six positional variables (e.g. montecarlo v_2//?vt*) defining position of a molecule in space (the so called pseudo-Brownian move).
   - ◊ change the loop conformation (e.g.

     ```
     set vrestraint a_/16:24
     montecarlo v_/16:24 local a_/16:24
     ```
2. local energy minimization;
3. calculation of the complete energy potentially including surface and advanced electrostatics terms (`REBEL` or `MIMEL`);
4. acceptance or rejection of this iteration based on the energy and the temperature.

Three possibilities for variable selections arguments:

- ♦ no variable selections: both *vs_MC* and *vs_minimize* will be set to all free variables. Some vs_MC variables, such as torsions rotating methyl groups, NH2 groups , will be automatically filtered out, since it is enough to just locally minimize them.
- ♦ one variable selection: the specified selection will be considered as the *vs_MC* , *vs_minimize* will be **the same** *vs_MC.*
- ♦ two variable selections: the first one is *vs_MC* selection, the second one is *vs_minimize.* **Important:** if two selections are explicitly specified, only *vs_MC* & *vs_minimize* will be set free. It means that during the montecarlo procedure the object will be fixed differently than before. After the command, the status of variables will be returned as they were before the montecarlo procedure. There are two basic possibilities: unfix on the fly or unfix first and then run montecarlo:

```
 montecarlo vs_MC vs_minimize # unfix on the fly
# OR
 unfix only vs_minimize # prepare fixation (vs_MC is a subset of vs_minimize)
 montecarlo vs_MC  # now one selection suffices and
                   # the object set of free variables is not changed
```

OPTIONS: `append`
appends to the existing conformational `stack` (overwrites by default).
`chiral`
temporarily activates the `l_racemicMC` variable

`tautomer`

toggles tautomer sampling. (`build-tautomer needs to be called before)

`fast :`
rapid side-chain optimization. This option allows one to accelerate the calculation by minimizing only a subset of the *strained* variables (as opposed to *all* minimization variables) after each step. The strain is established on the basis of the norm of the energy gradient after a random move. The strained variables are temporarily unfixed and this set of variables may be different every time. This option needs the `selectMinGrad` parameter to be set to about 1.5 (a threshold for the derivative norm). If this value is too low too many variables will be free and the procedure will be comparable with the default (non-fast) mode, if the parameter is too high the procedure may not be able to find the low energy conformations because the environment will not respond to the changes properly. Example:

```
build string "se ala his trp glu"
selectMinGrad=1.5
set vrestraint a_/*
montecarlo fast v_//x*
```

This mode is useful for side chain optimization in homology modeling.

`bfactor :`
you can use the `bfactor` option to sample 'hot' parts of structure with higher probabilities. The relative frequencies are taken from the b-factors of the atoms belonging to the mc-variables. Example:

```
build string "se ala his trp glu"  # default b-factor=20
set bfactor a_/2  1000.     # make 2nd his hot
montecarlo bfactor
```

To preserve the old bfactors, save them before the simulation and restore after. E.g.

```
b_old = Bfactor(a_//*)    # save
..
set bfactor a_/10:20 200.
montecarlo bfactor
..
set bfactor a_//* b_old  # restore
```

`local`
`local` [`dash`] [`a_/residueRange1,residueRange2...`]
(this option is specified after the main variable selections [*vs_MC* [*vs_minimize*]]) option `local` makes local deformation type movement for specified regions (e.g. two loops a_/15:22,41:55). Sub-option `dash` chooses angles for random deformation symmetrically with respect to the loop center. Note, that to avoid movements of the flanking regions around the loop, you need to set tethers for those regions. The local deformation only applies to the initial random

move, but the subsequence local energy minimization may move the flaking areas (in particular to the C-terminus side) away from their correct positions. The simplest way to set the tethers for the flanking residues (40:45 in the example below) is the following:

```
copy a_ tether      # create a copy of your current object
                    # and tether all atoms the original positions
delete tether a_//h* | a_/40:45
set terms "tz"      # add "tz" to the list of terms
montecarlo v_/40:45 local a_/40:45
```

`mute`
suppresses the text output about each random move

`output`
shortens the output by printing out only the steps with the **DY** (down/yes) outcome. The steps in which any of the simulation limits is reached are also shown. This option may considerably shorten log files of very long simulations.

r_exitEnergy real argument determines if you want your procedure to exit upon achievement of equal or lower energy value . For example, if you know energy of the minimum, you may want to stop the search when this value is achieved. E.g.

```
 build string IcmSequence("AHWEND")   # hexapeptide
 set vrestraint a_/* # BPMC-probability zones
 montecarlo 10. # stop after energy of 10. is reached
```

two atom selections: montecarlo .. as_1 as_2
(this option is NOT recommended for beginners) Atom selection arguments [*as_select1* [*as_select2*]] impose a filter on atom pairs considered in the terms of internal energy like "vw,el,hb,sf". There are three possibilities:

- ♦ *no selections* - the whole object (all atoms) is considered (the default)
- ♦ *as_select* - interactions of the specified atoms with ALL atoms in the object.
- ♦ *as_select1 as_select2* - interactions between two selections. For example, `a_dom1 a_dom1` would consider only the internal energy of the domain `dom1`.

`reverse`
this option makes a more intelligent random move in singlechain or a multichain molecule. By default if an angle is randomly changed near the beginning of a molecule, the second part of this chain moves. With the `reverse` the random move can occur in such a way that a part of the chain *above* a randomly chosen angle will stay the same, while the chain *below* the angle will move. Actually, the parts will be compared by molecular mass and the heavier part will be more likely to stay where it is than the lighter part. The probability that a part stays static is proportional to the number of atoms of this part. It is important that the virtual variables (`v_//?vt*` are not fixed).
This option is very useful in docking, since the receptor is static and the moving molecule should try to preserve the majority of current interactions. Also, the `reverse` option helps if one simulates the N-terminus of a multi-chain protein, or a docking of a peptide to a protein. Example:

```
 read pdb "1aya"  # read a complex
 delete a_!1,2    # keep only SH2 domain and a peptide
 convert          # make an ICM object with hydrogen
 set vrestraint a_/*         # set prob. zones
 montecarlo reverse v_2 v_2  # re-dock the peptide
```

If you move the **1st** molecule, do not forget to unfix the `fvt1` variables of all other molecules, e.g.

```
 ..
 unfix only v_1 | v_*//fvt1
 montecarlo reverse
```

If you always want to keep the C-terminus static and move the N-terminus, use the `superimpose` option (see below).

`store`

option `store` means that at the end of the simulation the stack is stored in the current object (equivalent to the `store stack object` command). This allows to extract it later without reading it from a file.

`superimpose` *as_3atoms_per_molecule*
superimposes new generated conformations after every move. Usually if you change backbone torsion at the N-terminus, the whole molecule moves. This option allows one to generate conformational changes at the N-terminal part of a peptide while its C-terminus occupies the same position in space. After each random move the first 3 atoms selected in molecule(s) will be superimposed on their initial position and the 6 positional variables (v_//?vt*) will be updated accordingly. The setup:

1. unselect the virtual variables from the MC selection (v_//!?vt*)
2. specify three or more atoms beyond the N-term. of interest for superposition
3. add virtual variables to the minimization selection (it is usually the default) to allow positional adjustments during minimization (the movements of C-terminus are suppressed only in the MC move, not in the following minimization).
4. if minimization is used (mncalls > 1), make a copy of the molecule and tether the C-terminus to it.

`trajectory`
records all accepted conformations sequentially in a binary `*.trj` file. Later one can `read trajectory`, `display trajectory`, and operate with individual frames, e.g.

```
for i=1,Nof(frames)
  load conf i       # to extract a frame
  display skin white center
  write image png "f"+i
endfor
```

Example:

```
 mncalls = 1 # move N-term residues a_/1:5 and while keeping
             # the rest in the same position
 montecarlo v_//!?vt* superimpose a_/6/c,ca,o
             # virtual variables should be available for minimization
 montecarlo v_/1:3/!omg,?vt* superimpose a_/6/c,ca,o
# Now a more realistic example
 build string "se ala his trp ala ala ala ala"
 display
 display residue label
 mncalls = 200
 copy a_1. "original"
 set tether a_/5:7 a_original./5:7
 set terms "tz"
 set vrestraint a_/*
 mncallsMC=100000
 montecarlo v_/1:4/!omg,?vt*  superimpose a_/5:7/ca
```

The following ICM-shell variables and commands are important for the procedure.

- `mncallsMC`,
- `mncalls`,
- `temperature`,
- tempCycle = {tempMax,tempMin,tempPeriod}, e.g. {1200.,600.,100000.} for a cyclic temperature schedule
- `mcBell` to make rs-zones narrower or wider than in `icm.res` file
- `mcJump`
- `mcShake` - the average amplitude of the pseudo-Brownian move
- `mcStep` - an amplitude of the unbiased step
- `l_bpmc` - if no, makes simple random steps (one angle by a random value)
- `l_writeStartObjMC` - if yes, write the starting object with its fixation and geometry to a file.
- `mnvisits` three limits and three actions follow
- `visitsAction` ,
- `mnhighEnergy` ,

- ♦ `highEnergyAction`,
- ♦ `mnreject`,
- ♦ `rejectAction`,
- ♦ `vicinity`,
- ♦ `compare`.

EXPLANATION OF THE OUTPUT (below are 3 example lines with numbered fields):

```
1     2       3   4     5     6   7     8     9         10          11   12      13    14
DY Visi     600  16   gln  xi3 70   -98    94   -322.04   -324.56   35    4.87 51559
__ __       600  32   ile BPMC ipt  ipt   ipt   -324.56   -290.80   18   65.72 51577
_Y Visi     600  16   gln BPMC qmm  qmt   qmm   -324.56   -323.93   41    3.78 51618
```

1. DY = Down Yes, i.e. energy has decreased after change and new conf. is accepted __ = up no , i.e. energy has increased and new conf. is not accepted _Y = up Yes, i.e. energy has increased, but new conf. is accepted
2. stack operation code indicates the outcome of comparison of the current conformation with the stack.
   - ◊ Impr : the conformation is close to one in the stack and has a better energy. Visited and improved
   - ◊ New : the conformation added as a new stack conformation
   - ◊ Sbst : not found, full stack, the worst is substituted for the current
   - ◊ Visi : visited and not improved
   - ◊ Vlm : visited and not improved, repetition limit mnvisits is achieved
   - ◊ High : not found, worse than the worst stack structure
   - ◊ __ : NO in calling routine (has nothing to do with stack)
   - ◊ RLim : NO limit of sequential Rejections is reached (has nothing to do with stack)
   - ◊ VLim : NO Vlm (number of visits > mnvisits)
   - ◊ HLim : NO High. mnHighEnergy limit is reached.
3. current temperature in Kelvin;
4. number of selected residue
5. selected residue name
6. name of randomly selected angle or BPMC to indicate the biased probability move
7. internal coordinate value or name of the multidimensional zone before random change;
8. internal coordinate value or name of the multidimensional zone after the random change but before minimization;
9. internal coordinate value or name of the multidimensional zone after the minimization;
10. energy before the random change;
11. energy after the random change and subsequent minimization;
12. number of function calls made during minimization;
13. gradient RMS deviation (normal completion is with low or zero gradient);
14. total number of function calls in the simulation.

The logic of stack operations is the following. There are three possible events for each slot of a stack:

1. new slot creation
2. energy improvement of the current slot conformational family
3. replacement of the looser conformational family by a better energy conformation

The starting conformation is placed to the first slot, if the stack is empty. At every simulation iteration, distances (either coordinate RMSD or angular RMSD, as defined by the `compare` command) are calculated between the current conformation and all slots. If any of the distances is less than the `vicinity` parameter, then the energies are compared and if the current conformation has the better energy, the stack conformation is replaced by the current one, otherwise the visit counter of the slot is incremented. If no similar structures are found, the conformation is appended to the stack, i.e. a new slot is created. If the stack is full, i.e. number of slots reached `mnconf` parameter, then the worst-energy structure will be substituted by the current, provided the latter has lower energy. Otherwise, no action is taken and *number_of_high_energy_conformation* counter is incremented (see also `mnhighEnergy`).
Explanation of the last section of the output. Example:

```
 Info> 4 stack conformations saved to def.cnf [3 compressed]
 Info> nSteps=     74, nTrials=      80, AcceptRatio= 0.92500,
 Info> BestEnergy=    -6.01, Step      37; nCalls=    2009, eachMcVar =  1.88
```

- ♦ the diverse low-energy stack conformations are saved in a very compact file. The `stack` can be later loaded with the `read stack`, `load conf` commands.
- ♦ `nSteps` - the number of accepted moves
- ♦ `nTrials` - the number of generated random moves
- ♦ `AcceptRatio` - nStep/nTrials

- ♦ `BestEnergy` - the best energy found by the stochastic optimizer.
- ♦ `nCalls` - the total number of energy evaluations (each random move includes multiple energy evaluation performed by the local minimizer)
- ♦ `eachMcVar` - the average number of attempts to change each variable (if this number is less than one, the sampling may be insufficient, also read about convergence).

---

move

Move objects, molecules between objects.

move ms_molecule: change tree topology

`move` *ms_moleculeToReconnect as_terminalAtom*
changes the topology of the basic ICM-tree by reconnecting the first `virtual` bond of a specified molecule to a given atom. This allows you to move two molecules together as one rigid body. By default, all the molecules are connected to the origin [0,0,0] through virtual bonds. The molecule can be connected only to the terminal atom, usually a hydrogen. The molecule can not be connected to itself (naturally, do not even try it). This operation is defined only for ICM molecular objects.
Examples:

```
build IcmSequence("AAFF;DEG")  # two molecules connected by virtual bonds
display virtual                # to the origin
move a_2 a_1/3/hz              # graft the second molecule to a hydrogen
                               # on another molecule
                               # now the second molecule will move together
                               # with the a_1/3/hz branch
                               # if you change v_1//?vt* variables
```

move : move multiple molecules between objects or merge two objects

`move` *ms_MoleculesToMove os_destination*

`move` *os_ObjectToMove os_destination*
move one, several or all selected molecules (*ms_MoleculesToMove* or *os_ObjectToMove)* to the specified object *os_destination.* When all the molecules are moved from the source object, the empty object is deleted. The *ms_MoleculesToMove* molecule or object are **appended** to the end of the *os_destination* object and their `virtual torsion` `tvt1` becomes virtual phase `fvt1`. This command is used to create one object from several components.
Examples:

```
read object s_icmhome+"crn"   # 1st object
build string "se ala his leu" # 2nd object
move a_2. a_1.                # take the 2nd obj and merge
                              # it with the 1st one
# Or
read pdb "1sis"
read pdb "2eti"
set object a_1.
move a_2. a_1.     # two PDB structures became one
                   # ICM molecular object
display virtual
```

move several molecules into any molecule, auto-bond several molecules

`move` *ms_molecules_to_merge* [*s_new_mol_name*]

`move only` *ms_molecules_to_merge*

different molecules in a PDB object can be merged into a single molecule and correct intermolecular bonds can be formed with this command. This command also automatically bonds the closest atoms (if distance< 0.6(R1+R2)) between the molecules being merged into a single molecules. Helpful in dealing with PDBs with disconnected carbohydrates. Options:

- ♦ If new name is not provided, the name is taken from the first molecule.
- ♦ `only` - do not merge, just make bonds between the closest atoms

Example;

```
read pdb "1nxc" # three parts of hetero-mol need to be merged
move a_2,3,4 "glycan"   # they are merged and bonded now.
move a_1,2  only        # form a bond with the protein
```

move a table row or parray element

`move` *table*[*i_row*] [*i_newPos*]

moves a row, e.g. t[2] to a new position. If the position is not specified the row is moved to the end.

`move` *parray*[*i_pos*] [*i_newPos*]

moves a `parray` element to a different position.

Example:

```
group table t {1 2 3}
move t[2] 1
```

move table column

`move` *table.column* [*i_newPos*]

moves the column to a new position. If the position is not specified the column is moved to the last position. Example:

```
add column t {1 2 3} {3 2 1}
move t.B 1
```

See also: `add column`, `add column function`

---

move alignment sequence

`move` *ali seq* [*i_new_seq_pos*]

move the sequence in an alignment to a new position. If the position is not specified, the sequence is moved to the last position. Example:

```
read alignment s_icmhome+"sh3"
move sh3 Eps8 1
move sh3 Fyn
```

See also: `Resorting alignment`

pause

`pause` [*i_n_seconds* | *r_seconds*] [*s_message*]
suspends execution for specified number of seconds. A fraction of a second can also be specified, e.g. `pause 0.01` If no argument is specified, the program will wait until RETURN is pressed. Examples:

```
 pause 0.1  # hundred milliseconds
 pause 5 "You have 5 secs"    # pause for 5 seconds

 read object s_icmhome+"dcLoop.ob"  # How to analyze the conformational stack
 read stack s_icmhome+"dcLoop.cnf"
 display a_//ca,c,n                 # display backbone
 for i=1,Nof(conf)                  # for all stack conformation
```

```
   load conf i                        # load and redisplay each of them
   pause "Press Return. N"+i          # gives you time to inspect the structure
 endfor                               # go on to the next conformation
```

Debugging shell scripts
The `pause` command also can set the program into a debugger mode in which you will be prompted to confirm each command by pressing RETURN. In the debugger mode the `l_commands` flag will be automatically set to `yes` and restored upon quitting. This is how to do it:

- To start the debugger mode, add to your script: `pause "START DEBUGGER"`
- To quit the debugger mode, type or add to your script: `pause "QUIT DEBUGGER"`

---

plot

create a PostScript file with a plot (for a built-in interactive plot use `make plot`, add `output=` *s_file.pdf* to save a pdf to a file).
`plot` { *R_Xdata R_Ydata* | *M_XmultpleYdata* } [*S_PointLabels*] [*S_PlotAxisTitles*] [{ *R_4Tics* | *R_8Tics* }] [*s_epsFileName*] [`options`]

- **Simple input:** Two compulsory arguments *R_Xdata R_Ydata* contain the X and Y coordinates. Both arrays may also be `integer arrays`.
- **Matrix input:** allows you to specify several data sets. The *M_XmultpleYdata* matrix may contain either X,Y1,Y2,..Yn columns or just Y1,Y2,..Yn columns if option `number` is used. Matrix M[2,n] or M[n,2] is equivalent to the simple input *R_Xdata R_Ydata* (Note that function `Histogram(`) returns such a matrix). Additional convenience: by default, different data sets will be shown in different colors and a panel with series/color correspondence will appear at the position specified by the `PLOT.seriesLabels` preference (choose "none" to suppress the panel). To avoid ambiguity do not use explicit *S_PointLabels* with the matrix input. Example:

```
 # table t . It has columnds t.A and t.B add column t Random(1. 5. 20) name="A"
add column t Random(1. 5. 20) name="B"

# now let us make a matrix with one column containing the order number
m=Transpose(Matrix(Rarray(Count(Nof(t))))) # 1. to 20. column in a matrix
m=m//Transpose(t.A) # add column A from t
m=m//Transpose(t.B)
# add your functions of X here

plot m display
# or
plot m square display
```

- **Axis and Tics:** 8-array *R_Tics[1:8]* contains information about X and Y axis: { *Xfrom, Xto, XmajorTics, XminorTics, Yfrom, Yto, YmajorTics, YminorTics* }. If only 4 numbers are provided, they are interpreted as { *Xfrom, Xto, XmajorTics, XminorTics* } while the Y axis tic marks are determined automatically. By default, if this argument is missing, the tic marks for both axes are calculated automatically. Example:

```
x={1. 3. 4. 7. 11. 18.}
y=Sqrt(x)
plot x y {0.,30.,2.,4.}               # only X-axis marks are defined
plot x y {0.,30.,2.,4.,0.,10.,1.,5.} # both axes are explicitly defined
```

- **Title and legends**: string array *S_PlotAxesTitles[1:3+NofSeries]* contains { "Title", "X title", "Y title" } in the simplest case. If multiple series are plotted using *M_XmultpleYdata* or **number** *M_multpleYdata* arguments, each series may be named with additional components of the array: { "Title", "X title", "Y title","Y1 title","Y2 title",..}.
- **Plot controls**: Optional *S_PointLabels* has the same number of elements as *R_Xdata* or *R_Ydata* and may contain either string to be displayed at the corresponding X Y point, or control information about marker type, color and size. The control string must start with underscore (_). To display both symbols and string labels, duplicate X and Y arrays (e.g. X//X, Y//Y) and supply the first S_PointLabel section with the symbol information and the second one with the string label information. Examples of string labels:

```
s={"1crn", "2ins", "1gpu", "3kgb","4fbr","6cia"}       # text labels: show as i
s={"_red SQUARE 0.4", "", "", "_green DIAMOND","",""}  # control labels
s={"_line" "" "" "_red line" "" "" "_blue line" "" ""} # control labels
```

The empty string tells the program to inherit all the settings for the previous point. Individual components of the string label are (i) color, (ii) mark type and (iii) mark size. Omitted components are not changed. Allowed colors: ICM_colors from icm.clr file which one can show with `show color` command. The `Color` (*R*) function will return a string array with suitable for `plot` color names mapped onto values. Allowed mark types: **line, cross, square, triangle, diamond, circle, star, dstar, bar, dot, SQUARE, TRIANGLE, DIAMOND, CIRCLE, STAR, DSTAR, BAR.** Uppercase words indicate filled marks.

Options.

- ♦ `append` - append the plot to an existing plot file.
- ♦ `display` - view the created postscript file with an external viewer defined by the `s_psViewer` variable.
- ♦ `grid`, or `grid="x"`, or `grid="y"` - draw grid at the major tics for the specified axis. Default: for both axes ("xy").
- ♦ `exact` - data points can reside exactly at a margin.
- ♦ `regression` - draw linear regression line.
- ♦ `frame` - draw NO frame around the plot (paradox isn't it? yeaah we are tricky).
- ♦ `origin` - make origin at (0,0) point.
- ♦ `link` - enforce 1:1 aspect ratio, equivalent to `PLOT.Yratio` = 1.0 .
- ♦ `comment=` *S_xyXYtext* - this option allows one to draw one line of text along all the four sides of the plot box. The string array may contain up to four strings {s_x,s_y,s_X,s_Y}:
    1. s_x: lower horizontal string, i.e. comment={"xxxxxxx"}
    2. s_y: left vertical string, i.e. comment={"","yyy"}
    3. s_X: upper horizontal string, i.e. comment={"","","XXXXXXX"}
    4. s_Y: right vertical string, i.e. comment={"x","y","X","YYY"}

  This option may be used to draw amino acid sequence around a contact plot box or a dot plot box.
- ♦ `number` generates the sequential numbering for X-array if this array is missing and sets a natural X tic style. In case of matrix input (see above) option `number` allows one to omit the X-array.

String variable *s_epsFileName* with extension `.eps` defines the name of a PostScript file where the resulting plot is to be written to. The default of *s_epsFileName* is `"def.eps"`.
Examples:

```
x = Rarray(90,0.,360.)             # an array of angles with 4 deg. steps
plot x Sin(x) display
plot x//x Sin(x)//Cos(x) display   # quick and dirty way to have two data sets.
                                     # Now let us get rid of the defect
s = Sarray(2*Nof(x))               # S_PointLabels for both arrays
s[Nof(x)+1] = "_red line"          # restart line for the first point
                                   # of the second set
plot x//x Sin(x)//Cos(x) s display # much better

plot Transpose(x)//Transpose(Sin(x))//Transpose(Cos(x)) display

read object s_icmhome+"crn"
crn_m = Sequence(a_/A)             # a_/A ignores termini
plot comment=String(crn_m)+Sstructure(a_/A) number Turn(crn_m) display # try it
plot comment=String(crn_m)+Sstructure(a_/A) number Turn(crn_m) {"Turn prediction","Res
unix gs def.eps                    # to see it again
```

See also: `make plot` , `Histogram`, `plotRama` macro in the `_macro` file, and examples in the `_demo_plot` file.

plot area: show matrix values with color

`plot area` *M_XYdata* `options` [*S_TitleXY*] [{ *R_4Tics* | *R_8Tics* }] [*s_epsFileName*]
plot 2D data from the matrix and mark values by color. Other arguments are the same as in the `plot` command. Distribution of colors is controlled by the `PLOT.rainbowStyle` preference. By default the minimal and maximal values of matrix *M_XYdata* are used as extremes for coloring. **Options**:

- ♦ `color=` *R_2MinMax* option allows you to enforce specific boundaries represented by the color range. For example, if you chose the "blue/red" PLOT.rainbowStyle the matrix value smaller than or equal to the first element of the *R_MinMax* array will be colored blue, while the matrix values larger than or equal to the second element of the array will be colored red, the middle values will be color with intermediate colors. The real array of boundaries contains two elements.

```
PLOT.rainbowStyle = "blue/white/red"
```

```
color={1. 3.}  # <= 1. are blue; above 3. red
color={3. 1.}  # >= 3. are blue; <= 1. are red
```

- ♦ `link` - enforce square shape (1:1 aspect ratio) of each cell, overrides `PLOT.Yratio`.
- ♦ `comment=` *S_xyXYtext* this option allows one to draw one line of text along all the four sides of the plot box. The string array may contain up to four strings {s_x,s_y,s_X,s_Y}:
    1. s_x: lower horizontal string, i.e. comment={"xxxxxxx"}
    2. s_y: left vertical string, i.e. comment={"","yyy"}
    3. s_X: upper horizontal string, i.e. comment={"","","XXXXXXX"}
    4. s_Y: right vertical string, i.e. comment={"x","y","X","YYY"}

  This option may be used to draw amino acid sequence around a contact plot box or a dot plot box.
- ♦ `transparent=` *R_2range* option allows you to make a certain range of matrix values invisible. If *R_2range[1] < R_2range[2]*, the specified range will be excluded from the plot, while the values beyond the range will be shown. If *R_2range[1] > R_2range[2]*, the specified range will be shown by color, while the values beyond the range will be excluded. Example:

```
transparent={1. 3.}  # values WITHIN  the range are not shown
transparent={3. 1.}  # values OUTSIDE the range are not shown
```

Data can also be transformed and clamped with the `Trim( )` function.
Examples:

```
 read matrix s_icmhome+"def.mat"
 PLOT.rainbowStyle = "blue/white/red"
 plot area def display  # min/max = {-3.,17.}
 plot area def color = { 0., 20.} display
 plot area def color={-0.,15.} transparent={-10.,5.} display
 plot area def[1:12,1:10] link display comment={"X","Y axis"}
#
#
 N=210
 M=Matrix(N N)
 for i=1,N
   M[i,?]=Sin((Power(i-12.1 2)+Power(Count(N)-12.1 2)))
 endfor
 plot area M link display
     # just a nice test, default boundaries are used

 read pdb "1crn"
 MDIST=Distance(Xyz(a_//ca))
 s=String(Sequence(a_1./A) )
 PLOT.rainbowStyle = "blue/rainbow/red"
                   # contact map for 1crn, values below 4.8 and
                   # above 10. A are not shown
 plot area MDIST area color = {4.5 15.} transparent={10.,4.8} \
        display link grid comment=s//s
```

See also the `make plot` associated plot method, for example

```
m = Matrix(10)
add header t m name="m"
make plot t "matrix=m;rainbow=white/yellow/green"
```

predict

`predict` *model T_n* [*M_nxm*] [key] [name= *s_colName*]

command applying a model developed by the `learn` command.

print

`print` *arg1 arg2 arg3 ...*

The arguments may be variables or constants of `integer`, `real`, `string`, `logical`, `iarray`, `rarray`, `sarray`, `matrix`, `sequence`, or `alignment` type.
Examples:

```
 print "no. of atoms=", i_out, "GRAPHICS.wormRadius=", GRAPHICS.wormRadius
```

print bar : showing progress bar from ICM shell

`print bar` { `"."` | `" Start"` | `"End\n"` } *nSteps*

Useful in showing progress in a long for loop. Example:

```
l_commands = no
print bar " Start" 100
for i=1,100
  read pdb "1crn"
  rm a_
  print bar "." 100
endfor
print bar " End\n"
```

printf

a family of three functions for the formatted print:

- printf s_formatString args ... # prints to stdout and s_out
- sprintf [append] s_formatString args ... # prints to s_out only
- fprintf [append] s_file s_formatString args ... # write to a file

`printf` *s_formatString arg1 arg1 arg2 arg3 ...*
formatted print, mostly follows the C-language printf syntax. The arguments may be variables or constants of only `integer`, `real`, `string` type.
s_formatString may contain

- **plain characters** that are directly reproduced
- **ambiguous characters**: \\ - backslash, \" - double quote, %% - percent
- **escape sequences** for more tricky characters (\a - bell, \b - backspace, \f - formfeed, \n - newline, \r - carriage return, \t - horizontal tab, \v - vertical tab) and
- **conversion specifications** for each argument of the printf command. Each specification starts from **%** and may be followed by **-** sign for left adjustment, and precision specification (e.g. %-**5**.2f).
    - ◊ %c - unsigned character
    - ◊ %s - string
    - ◊ %d %D - integer
    - ◊ %[-] *i1.i2f* - float (real) in decimal notation
    - ◊ %g %G - real in either f or e style, precision specifies the number of significant digits.
    - ◊ %e %E - real in [-]d.ddde+dd style
    - ◊ %o %O - unsigned octal
    - ◊ %u %U - unsigned decimal
    - ◊ %x %X - unsigned hexadecimal

The output is directed to the screen and is also saved in the `s_out` string which can be later written or appended to a file.
Examples:

```
 printf "Resol. = %4.1f N_ml= %-3d\n", a, n
 write append s_out "log"                   # append to the log file
```

See also: `sprintf [append] [s_]` (prints to the `s_out` string by default) `fprintf [append] s_file` (directly prints to a file).

print image

`print image` [`window=` *I_xyPixelSizes*]
print the current screen image to the printer defined by the `s_printCommand` ICM string variable. Use option `window=` to increase the resolution (however in this case bear in mind that the lines will get thinner and labels smaller). Be kind to your printer and color the background white (e.g. `Ctrl-E`). See also: `write image` `s_printCommand`, `View` (`window)` **.**
Example:

```
 read pdb "4fgf"
 nice "4fgf"
 color background white # or press Ctrl-E
 print image
# or
 s_printCommand  = "lp -c -ddepartmentalColorPrinter"
 print image window=View(window)*2 # increase resolution two-fold
```

Run SQL queries

`query molcart` *s_sql_command*|*S_sql_commands* [`name=`*s_tableName*] [*connection_options*]

Performs an SQL query in the connection specified by *`connection_options`*. For **SELECT** and other queries returning data, this command creates a table. The result table may be specified by the *s_tableName* parameter. All SQL types are converted to appropriate ICM types.

Example:

```
 query molcart "select * from asgsynth where molid=1"
```

See also: `molcart`, `find molcart`, `load molcart`

quit

`quit` [*s_message1 s_message2* ..]
Terminates ICM session. Note that the message strings can not contain expressions or functions. Example using two strings:

```
  HELP = " $P - program to do things"
  if Getarg()!="" quit " unrecognized arguments. " HELP
```

randomize

a group of commands to modify ICM objects using random numbers.

randomize internal variables in molecules

`randomize` *vs r_angAmplitude*
randomly distort current values of specified variables with either specified or default amplitude in degrees for angles and in Angstroms for bonds. The range is [CurrentValue - *r_angAmplitude*, CurrentValue + *r_angAmplitude*]. Default amplitude is defined by `mcJump` ICM-shell variable (30.0).
randomize variables in range
`randomize` *vs r_angMin*, *r_angMax*
assigns random values within specified range to selected variables.
randomize atom positions
`randomize` *as r_amplitude*
translates the specified atoms `as_` randomly and isotropically according to Gaussian distribution with the specified sigma.
randomize molecule positions
`randomize` *ms r_amplitude*
translates and rotates the specified molecules *`ms_`* randomly and isotropically according to Gaussian distribution with the specified sigma. We call it a Pseudo-Brownian random move. The same moves are used in the montecarlo docking protocol.
Examples:

```
 build string "se ala glu tyr"
 randomize v_//!omg 50.   # distort all variables with
                          # 50 degrees amplitude

 randomize v_/14:21/phi,PSI -70., -50. # range [-70.,-50.]

 copy a_ "ttt"
 mv a_ttt. a_
 randomize a_2

 randomize a_/tyr/!ca,c,n,o 0.05
```

(Note use of `PSI` torsion in the last example.)

read

read stuff from a disk file, pipe or string.
ICM offers several ways of reading information in:

read from file

`read` ... *s_fileName* [`mute`] [`pattern=` *regexp*]
reading from a file. Just specify the type and from what file. The file name is a string and must be quoted. Usually, the extension can be omitted if it is standard and is implied by the object type. Also, in several cases the program will try to find the requested file in a special directory (`s_pdbDir` for a PDB file, `s_xpdbDir` for an xpdb object, etc.), if is not found in the current one.
Option `mute` will temporarily switch `l_info` to `no` .

Option `pattern` = *regexp* will filter out the lines of the text file that match the regular expression.

Examples:

```
 read pdb "1crn"
      # s_pdbDir will also be searched.
      # It will also read "1crn.brk.Z"

      # you may specify file extension explicitly
 read iarray "a.a" mute
```

read binary and read binary list

`read` `binary` [`name=` *S_objNames* [`class1 class2` ..]] [*s_fileName*] [`mute`] [`display` | `only` | `all`] [`edit`] [`list`]

`read` `binary` pdb *s_pdb_code* # see also s_xpdbDir
reads icm-portable binary project file. ICM allows one to save multiple ICM-shell objects to a single compact cross-platform binary file.

Reading everything or just some items By default, ICM reads all objects in the file. If you want to see the list the objects in this archive, use the `list binary` command. To read **selectively**, use the `name` option or a list of object classes, e.g.

```
read binary object alignment name={"a","b"} "a.icb"
# only extracts 3D objects, alignments, and variables named a and b
```

Options:

`name=` *S_objNames*|*name={"g1","m_gb"}*|*reads* a subset of objects	
`mute` | `read` binary mute	temporarily switches `l_info` to `no`
`display` | `read` binary display	displays molecules as they were saved
`only` | `read` binary udisplay	ignore display section, **only** read
`undisplay` | `read` binary only	same as `only`
`list` | `read` binary list	makes a table of content for the file
`edit` | `read` binary all edit *s_file*| reads password protected files	

`read` `binary` `list` [`name` = *s_outputTableName*] *s_fileName*

creates **a table of content** for the icm objects stored in a binary file. It the table name is not specified, `T_out` table is created. From the GUI interface you can double click on a table row to download a particular object from the file.

```
 read binary only    # reads the default icm.icb file and suppress display
 read binary "aaa"   # reads all objects from aaa.icb
 read binary name={"biotin","DOCK1_rec"} "example_docking"
 read binary "example_docking" display  # reads and displays as saved

 read binary list "example_docking" name="ed_toc"
 read binary edit "secretfile.icb"  # will be prompted for the password

 s_xpdbDir = "http://ablab.ucsd.edu/xpdb/"
 read binary pdb "1xbb"
```

read html file

`read html` [`display` | `auto`] *s_htmlFile* [`name=` *s_newStringName*]

read an html file and display its contents in the built-in ICM html-browser. This command also creates a string variable. Options:

- ♦ `name=` *s_newStringName* : gives the ICM variable a name (the file name root is the default)
- ♦ `display` - create a string variable with the file contents and display the file
- ♦ `auto` - makes the document pop in `txdoc` every time you read a project with this string.

If a html-document is read to ICM with this command, it can be stored in a single project along with other objects of the ICM shell.

`read html simple` *s_htmlFile*

does not create a shell variable, just displays the file with `txdoc` .

The HTML documents can contain sections of the icm code (so called `icmscript`) which are executed upon clicking. Example:

```
<!--icmscript name="part1"
read pdb "1crn"
display a_*.
-->
...
<a name="part1" href="#part1">click here to execute icm script</a>
```

See also: `help browser` .

read from string

`read` ... `input=` *s_bufferString* [`name=` *s_newName*]
reading from an ICM string. Replacing file by a string is useful in CGI scripts, because the input information is easily accessible as an ICM string. Option `name=` *s_newName* allows one to specify a name of the new ICM-shell object. Note that multiline input can be directly pasted or typed after a triple quote followed by the closing triple quote.
Examples:

```
 s_mat="1 2\n3 5\n0 6"
 read matrix input=s_mat name="m23" # matrix m23 is created
 s_seq = "> a\nAFSGFASG\n> b\nQRWTERQWTE\n"
 read sequence input=s_seq  # read sequences a and b
 show a b
#
# using triple quoted multiline input:
  read matrix name='z' input="""
1 2 3
2 3 4
5 6 7
"""
```

read through filters: assign action by file extension.

`read` ... *s_compressed_or_encoded_files*
of any type directly. The files will be uncompressed on the fly, if the file extension and the corresponding filtering command are found in the the `FILTER` table. ICM understands `.gz` (`gzip`), `.bz2` (`bzip2`) and `.Z` (`compress`) compression.
Examples:

```
 read object "aa.ob.gz"
 read pdb "/data/pdb/pdb1crn.ent.Z"
```

read all

`read all` *s_allFileName*
reading from a `mixed file` containing several ICM-shell objects (including tables) or data types. Legal types and separators:

- #>i `integer_name`
- #>r `real_name`
- #>s `string_name`
- #>l `logical_name`
- #>p `preference_name`
- #>I `iarray_name`
- #>R `rarray_name`
- #>S `sarray_name`
- #>M `matrix_name`
- #>seq `sequence_name`
- #>prf `profile_name`
- #>ali `alignment_name`
- #>m `map_name`
- #>g `grob_name`

- #>T `table_name` # the column layout
- #>col `table_name` # the column layout
- #>db `table_name` # the database layout

- #> `brk` # a protein-data-bank file content
- #> `var` # internal variables (torsions, angles, bonds) for the current ICM-object

Example:

```
 read all "a.all"  # the file is given below
```

The `a.all` file may look like this:

```
#>r lineWidth
  1.00
#>R box4
 0. 0. 1. 1.
#>s tt.h
this is a header string of table tt. The arrays follow.
#>i tt.n
15
#>T tt
#> name bd nlines
icm 1985 160000
bee 1998 100000
inet 2000 80000
```

Such a file can be created with the
`write` append *icmShellObject file.all*
command

reading records from a large file via index table

`read` { `sequence` | `mol` | `mol2` } *T_selectedEntries*
extract database entries selected via index `table expression`. A large file with multiple records (e.g. an `.sdf` file, an `.ml2` file, a `.fasta` file, etc.) can be indexed with the `write index` command and then individual records or groups of records can be read via this index. The entries can also be extracted into a string array via the `Sarray`(*T_selectedEntries*) function.

Example:

```
 read index "/data/inx/SWISS.inx"
 read sequence SWISS[2:15]
 read sequence SWISS.ID ~ "IL2_*" | SWISS.ID == "ML2_HUMAN"
 # or
 read index "NCI3D"
 read mol2 NCI3D.DE ~ "^benz*"
 sarray_of_ml2s = Sarray( NCI3D[1:10] )
```

See the `readMolNames` sarray for details on database compound name storage conventions. Index file contains an integer position of the first character of an entry (ST as in STart), and the entry length (LE as in LEngth). Accepted types of the database index files are single files with multiple entries:

```
#>s Swiss.DIR
/data/swissprot/seq
#>s Swiss.EXT
.dat
#>T Swiss
#>--ID---------ST-------DA------LE-
104K_THEPA      0       906     1094
..
```

See also:

- ♦ `write index`
- ♦ `Sarray index`

read http/ftp

`read` .. "`ftp://ftp.server.com/path/to/file`"

reading directly from **ftp** port.

The ICM can read not only from files directly accessible from your computer but also files from remote locations via ftp or http.

ICM includes a simple FTP client to simplify access to the databases on the internet. Files names may be specified as an ftp style URL:
`ftp://` [*user* [: *password*]@] *hostname* [: *port*]/ *path/* file
If the password portion is omitted, the password will be prompted for. If both the user and password are omitted, anonymous ftp is used. In all cases passive (PASV) ftp transfers are used. If port is omitted, standard port (:21) is used.
Example:

```
 read binary "ftp://hestia.sgc.ox.ac.uk/pub/datapacks/CENTG1_annot_NEW.icb"
 read sarray "ftp://ftp.rcsb.org/pub/pdb/data/structures/divided/pdb/"+\
             "ab/pdb1ab1.ent.Z"
 read sequence "ftp://embl-heidelberg.de/toby/ph.seq"
```

URL-header may be used in existing mechanism of access to PDB:

```
 s_pdbDir ="ftp://ftp.rcsb.org/pub/pdb/data/structures/divided/pdb/"
 pdbDirStyle = "ab/pdb1abc.ent.Z"
 read pdb "1crn"
```

Remote files are stored in your local `s_tempDir` directory. Do not forget to delete them from time to time. The system table `FTP` can be configured to delete temporary files and deal with firewalls.

`read` .. "`http://www.server.com/path/to/file`" reading directly from **http** port.

ICM includes a simple HTTP client to simplify access to the databases on the internet. Files names may be specified as an http style URL:
`http://` [*user* [: *password*]@] *hostname* [: *port*]/ *path/* file[(?|
)name1=value1&name2=value2...]

Example:

```
read binary "http://hestia.sgc.ox.ac.uk/pub/datapacks/CENTG1_annot_NEW.icb"
read pdb "http://www.pdb.bnl.gov/pdb-bin/send-pdb?id=1crn"
```

You may pass arguments to the http URL using POST or GET methods.

- For GET method add '?' followed by URL encoded string in 'name=value&name=value' format. Use `String( s_string html )` to URL encode parameters Example:

```
read string "http://www.google.com/search?hl=en&q=molsoft&btnG=Search"
# if you query contains spaces or other non alpha-numeric characters you must UR
read string "http://www.google.com/search?hl=en&q=" + String("molsoft icm") + "&
```

- POST method differs from GET only by replacing '?' with a single ' ' (space). This method is widely used when communicating with `SOAP` services. Example script allowing to make a spelling suggestion:

```
url = "http://api.google.com/search/beta2"
HTTP.postContentType = "text/xml"
HTTP.protocolVersion = "1.0"

# form SOAP message

# create a message with SOAP method and a namespace
req = SoapMessage( "doSpellingSuggestion","urn:GoogleSearch" )
# add method arguments
req = SoapMessage( req,  "key","btnHoYxQFHKZvePMa/onfB2tXKBJisej" ) # get key fr
req = SoapMessage( req,  "pharse", "Bretney Spers" )  # some misspelled pharse

# send it to the server and read the resulk
read string url + " " + String( req )

# parse result
res = SoapMessage( s_out )

# check for errors
if Error(res) != "" then
  print Error(res)
else
  print Value(res)
endif
```

See also: `HTTP.proxy` `FTP.proxy`

read unix

`read` .. `unix` `unix_command`
reading from a unix pipe. (Note that you can read unix shell variables directly with the `Getenv(` *s_varName)*} function).
Examples:

```
read unix date
if(s_out[1:3]=="Sun")print "Go to church"

read column unix grep "^DY" f1.ou | awk '{print $11, $12}'
show def
```

read unix cat

`read` .. `unix` `cat`
reading from a buffer pasted with the mouse is a special case of reading from a unix pipe. Basically, just mark anything ICM-readable in any window, paste it to your ICM session and press `Ctrl-D`. Note that a file name which is usually used to name the ICM-shell object is missing now, therefore it may be named 'def' (i.e. default), rename it afterwards.
Examples:

```
  read alignment unix cat
  cd59n LQCYNCPNP--TADCKTAVNCSSDFDACLITKAG--------LQVYNKCWK
  ly6n  LECYQCYGVPFETSCP-SITCPYPDGVCVTQEAAVIVDSQTRKVKNNLCLP
  ^D
  show def
  rename def cd_ly

  read sequence unix cat
> cd59
  LQCYNCPNPTADCKTAVNCSSDFDACLITKAG
  LQVYNKCWKFEHCNFNDVTTRLRENELTYYCCKKDLCNFNEQLEN
  ^D

# read unix cat or read string are two equivalent ways to
# load text to the  s_out string
  read string
  This is the text which will end up
  in you s_out string.
  ^D

# read a mixed, read all -type, input and create two ICM-shell variables:
read all unix cat
#>s ss
strrr
#>i aa
234
^D
```

read alignment

`read alignment` [`fasta` | `pir` | `msf`] [*s_aliFileNameRoot*] [`name=` *s_aliName*]
read alignment file in a natural, pir or msf `formats`. Upon reading, all the sequences are created as separate ICM-shell objects. The alignment is created as a separate object for msf-formatted files. In the case of other formats the alignment object is created if lengths of all the sequences together with dashed ("---") insertions are equal to each other.

read color

`read color` *s_clrFile*
If you want to have an alternative color file (say, "icmw.clr"), you can reread the colors.
Example:

```
 read color "icmw"
```

read comp_matrix

`read comp_matrix` [*s_cmpFileNameRoot*]
reads cmp-formatted file (`*.cmp`) containing one or several residue comparison matrices.

read conf: conformations from file

`read conf` [*i_stackConf*] [*s_stackFileNameRoot*]
reads and sets one specified conformation from the conformational stack file `*.cnf`. If *i_stackConf* is omitted the best energy conformation is extracted. This command will work with both compressed and uncompressed (old) stack file formats.
See also `read stack`.

read csd

`read csd` [*s_csdFileNameRoot* [*s_csdJournalFileName*]] [*i_NofObjectsLimit* [*i_startingObject*]]
reads the output of the Cambridge Structural Database (CSD) `search utility`, namely, FDAT-formatted file (*.dat) and the optional session journal-file (*.jnl). Information about atomic coordinates, connectivity, parameters and symmetry of crystallographic cell is taken from the FDAT file. The journal file contains information about chemical names of compounds. If not

provided, the REFCODE csd-name is assigned to the compound name of the ICM-object. (See also Name ([*os_* ,] real)). Optional *i_NofObjectsLimit* and *i_startingObject* arguments allow you to extract a subset of several objects from a certain position of a multi-entry file. You can loop through all the objects by reading the chunks of up to about 1000 objects by doing the following:

```
 offset = 1
 while( yes )                 # infinite loop
   read csd "large" 100 offset  # read the next 100 objects
   if(Nof(object) == 0 ) break  # exit upon reading all obj.
#
#    do whatever you want
#
   offset = offset + 100
   delete a_*.
 endwhile
```

The object created is not of the ICM-type, use convert or write library to create an object or an ICM-library entry, respectively. Note that you can also read compressed CSD files (see FILTER).
Examples:

```
          # all objects from ex_csd.dat and ex_csd.jnl
 read csd "ex_csd"
          # only the first obj. ; explicit name for the journal file
 read csd "ex_csd" "ex_csd" 1
```

To see how to generate all the symmetry-related molecules in the cell, see the transform command.

read database

read database [field= *S_fields*] [group [name= *s_tableName*]] *s_databaseFileName*
read a text database with strings and numbers and create appropriate arrays. The field names in the database become names of the arrays upon reading. The list of array names will be stored in s_out . Option group indicates that a table should be formed (or ICM-shell structure) of the constituent arrays. This table will be renamed if option name is specified.
You may also group arrays of the database to form a table with a separate command. That will allow you to sort all the arrays and search all the fields by the Find( ) function.
Examples:

```
 read database field ={"NA","RZ"} s_icmhome+"foldbank.db" group name="tt"
 read database field ={"RZ","NA"} s_icmhome+"foldbank.db" group
 show foldbank
#
# ANOTHER EXAMPLE
 read database "LIST.db"
 show database $s_out  # you may also list the arrays explicitly
 write database $s_out "out.db"
```

See also: read column, write database, show database.

read drestraint

read drestraint [only][*s_cnFileNameRoot*]
read distance restraints (often referred to as *cn*) from an a .cn file. Do not forget to read drestraint types first. Option only tells the program to delete previous distance restraint settings.

read drestraint type

read drestraint type [only] [*s_cntFileNameRoot*]
read distance restraint types from a *.cnt file. Option only tells the program to delete all previous distance restraint types settings.

read factor

`read factor` [*s_factorFileNameRoot*]
reads the Xplor-formatted structure factor file. The input is free-field, and each reflection record may be extended over several lines.
Example:

```
INDEx 1 2 3  FOBS=9.0   SIGMA=3.3  Phase=50.0   Fom=0.8
INDEx 2 -3 1 FOBS=31.0   SIGMA=2.3  Phase=20.0   Fom=0.3
INDEx 5 6 6  FOBS=44.0   SIGMA=2.0
```

To read the ICM-formatted structure factor table, just use the `read table` command. ICM will recognize the file type.

---

read gamess from the output file.

`read gamess` *s_gamessOutputFile*

reads and parses the output of the `gamess` program. ICM converts the atomic (Hartree) energy units into `kcal/mole` and with some options can upload the minimized conformation.

read grob

`read grob` *s_groFileNameRoot* [`name=`*s_grobname*]
read `graphics object` from a `file`. If the `name` is not specified, The object name is derived from the file name. This command supports various import formats:

- ♦ Simple ICM graphics object format: ".gro"
- ♦ Wavefront OBJ: ".obj"
- ♦ OFF (Object File Format): ".off" , the default
- ♦ Google Earth KMZ: ".kmz"
- ♦ COLLADA: ".dae"
- ♦ 3DXML: ".3dxml"

Examples:

```
 read grob s_icmhome+"/icos"  # load icosahedron from icos.gro file
 display icos   # the name derived from the file name
 read grob s_icmhome+"/cube.gro" name="g"
 display g
 read grob s_icmhome+"/squirrel.kmz"
 display squirrel
```

See also: `write grob` .

---

read iarray

`read iarray` *s_iarrayFileName* [`name=` *s_newIarrayName*]
read `integer array` from a file. File `format` is free.

---

read index

`read index` *s_indexTableFile* [`name=` *s_ixFileName*] [`database=` *s_newDataBaseDirectoryName*]
read the index file for quick access to a database. The optional argument allows one to access the database file at a location different from those specified in the course of indexing with the `write index` command.
Examples:

```
 group table NCBI_ {"ID","DE","SQ"} "fd" \
        header "/data/nr/" "DIR" {"nr"} "FI" "" "EXT"
# we created control table t
  write index fasta NBCI_ "/data/nr/NR.inx"
                   # make index and save to a file
  read index "/data/icm/inx/NR.inx"
                   # read index
  show NR[2:5]
                   # usage of the last optional argument
                   # move the data file, keep the index file
  unix mv /data/nr/nr /newdisk/data1/nr/nr
  read index "/data/icm/inx/NR.inx" database="/newdisk/data1/nr/nr"
```

read library

`read library` [*s_libraryFileNameRoot*]

`read library` [`residue` | `atom` | `color` | `drestraint` | `vrestraint` | `charge` | `energy`] [*s_libraryFileNameRoot*]
reads the ICM library files:

- `icm.res` and user residue libraries. Several residue libraries can be used. The `LIBRARY.res` string array defines the residue library files which are loaded into ICM. For example, to add your library file `jack.res` to the existing residue libraries, you can do the following:

  ```
  LIBRARY.res = LIBRARY.res // "/home/jack/jack.res"
  read library
  ```
- `icm.bbt` - bond bend angle bending and improper torsion deformation parameters
- `icm.bst` - bond stretching parameters
- `icm.cod` - atom codes and types
- `icm.tot` - torsion angle energy parameters
- `icm.hbt` - hydrogen bonding parameters
- `icm.hdt` - surface-based hydration parameters
- `icm.cmp` - residue comparison matrix(es)
- `icm.cnt` - distance restraint types (cn)
- `icm.vwt` - van der Waals energy parameters (keyword `energy`)
- `icm.rst` - multidimensional variable restraints zones

The default library path is defined by the `s_icmhome` variable and the name is defined by the `s_lib` string ICM-shell variable.
Examples:

```
read library   # reads all library files according to LIBRARY table
LIBRARY.res = {"icm","/home/jack/jack.res"}
read library residue            # to re-read only residue libraries
read library atom "new.cod"     # re-read different atom codes
read library color "new.clr"    # different colors
read library drestraint "new.cnt" # drestraint types
read library vrestraint "new.rst" # vrestraint types
read library charge "new.bci"     # charge increments
```

read library mmff

`read library mmff` [*s_libraryFileNameRoot*]
reads the following additional library files for the `mmff94` force field:

- mmff.bbt
- mmff.bst
- mmff.tor
- mmff.tot
- mmff.vwt

To calculate the `mmff` energy one needs to assign `atom types`, and `charges`. The force field is switched with the `ffMethod` preference. An example:
Example:

```
build string "se nter his cooh"
read library mmff
set type mmff
set charge mmff
display
minimize cartesian
```

read map

`read map` [`reverse` | `xplor`] [*s_mapFileNameRoot*] [`name=` *s_mapName*]
read ICM-electron-density `map file` and create an ICM-shell variable of the `map` type. ICM understands the following map formats:

- ♦ CCP4 binary maps
- ♦ Xplor text format

If you read an external binary map file in CCP4 format, ICM will automatically recognize the *Endian* (the order of bits in numbers) and perform the conversion required. Option *reverse forcibly changes the *Endian* for binary maps generated outside ICM under a different operating system. We can not support many other popular map formats, or sub-types of the CCP4 or Xplor formats generated by different program. Use the **mapman** program (Kleywegt, G.J. and Jones, T.A. (1996). xdlMAPMAN and xdlDATAMAN - programs for reformatting, analysis and manipulation of biomacromolecular electron-density maps and reflection data sets. Acta Cryst D52, 826-828) to reformat the map to one of the two supported formats if necessary. **Reading many maps at once**
`read map` [`reverse`] [*s_mapFileNames*] [`name=` *s_mapNames*]
read multiple files specified in comma-separated string (e.g. `"./map/gc`,./map/ge") and rename the maps by matching names from a comma-separated string. Examples:

```
read map "gc1,ge1,gh1" name="m_gc,m_ge,m_gh"

read map "./gc1,./map/ge1,./gh1" name="m_gc,m_ge,m_gh"
```

---

read matrix

`read matrix` [*s_matrixFileNameRoot*] [`name=` *s_MName*]
read ICM-matrix `file` and create an ICM-shell variable of the `matrix` type.

---

read mol

`read mol` *s_FileNameRoot*|*X_chemarray*
[`delete`|`auto`|`bond`|`simple`|`charge`|`type`|`stack`] [`number=` {*i_number*|*I_from_to*}]
[`name=`*s_rootName*]
read multi-molecule MDL `mol` -file (a.k.a. SD-file) or directly from a `chemical array` and create stripped molecular objects (they need further `conversion`). The molecules are named according to the first line of the name section of the mol/sd format. If this line is empty, the root name is taken from the option `name=` *s_rootName*, and the molecules are named like this: `"xx","xx2","xx3","xx4" ..` if the *s_rootName* is `"xx"` . If none provided the molecules are named 'm', 'm2', 'm3',..., sequentially. Note that with the `name` option the first molecule keeps the name exactly as specified in the `name` option. If possible `readMolNames` is utilized.
In the default mode a pattern of single and double bonds is interpreted in order to identify aromatic systems. Then appropriate bond types are changed to aromatic (hit `Ctrl-W` to see the effect). This aromatic system assignment, however, is irreversible. If you `write mol` after that the new bond types will be saved.
Set `l_readMolArom` to `no` if you do not want to assign aromatic rings upon reading. (and formal charge and bond symmetrization for CO2, SO2, NO2or3, PO3). To suppress suppress the symmetrization and consequential charging of CO2, set the `l_neutralAcids` to `yes` .
S_out contains all properties: All the property fields specified in the mol file, e.g.

```
<logp>
2.344
<cas>
234
```

will be stored in the `S_out` array (one string for each object). The string can be further split into fields to extract the values, e.g.

```
 cas  = Trim(Field(S_out,"cas_rn",1,"\n")) # sarray of cas numbers
 logp = Rarray(Field(S_out,"logp",1,"\n")) # rarray of logp values
```

Do not forget that ICM converts all strings to low-case.
Options:

- `exact`: enforces the exact mol/sd format. The default reading mode is more tolerant to common format violations.
- `auto`: automatically assigns compound names, if the name line is missing. The name is composed of the file name root and the order number of a compound.
- `hydrogen`: automatically adds hydrogens
- `type`: automatically assigns MMFF atom types
- `charge`: automatically assigns MMFF atom charges according to the types
- `stack`: read multiple conformations of into an object stack instead of reading them as separate objects

Examples:

```
 read mol "ex_mol.mol"  # you may skip the extension
 logP = Rarray(Trim(Field(S_out,"logp",1,"\n"))) # rarray of LogP values
 build hydrogen
 wireStyle="chemistry"
 display a_
```

Conformational generation The `_confGen` script creates a table called conformers. In this table multiple conformations of the same molecule can be recognized by column MOL_NUM with the molecular number in the input file. Now the multiple conformation can be read into a molecular object with a stack like this:

```
read mol stack (conformers.MOL_NUM == 2).mol # read molecule #2 grouping all conformati
```

If you do not need to create molecular objects, but need to create a molecular spreadsheet instead, use the `read table mol` command.
See also:

- `l_readMolArom`,
- `l_neutralAcids`,
- `read table mol`,
- `String` # e.g. read mol input=String(t.mol)

read mol2

`read mol2` [*s_FileNameRoot*]
read Tripos' Sybyl `mol2` -formatted file (extension .ml2) and create stripped molecular objects (they need further `conversion` to become ICM-objects).
Set `l_readMolArom` to `no` if you do not want to assign aromatic rings upon reading. (and formal charge and bond symmetrization for CO2, SO2, NO2or3, PO3). To suppress suppress the symmetrization and consequential charging of the acidic groups like CO2, SO3, PO3 set the `l_neutralAcids` to `yes` . These will work only if the input files contain only single and double bonds (no aromatic types).
Examples:

```
 read mol2 "ex_mol2"  # this example file is provided
```

read trajectory

`read trajectory` [*s_movFileNameRoot*]
read ICM-trajectory `file` with the Monte Carlo simulation trajectory.
See also: `display trajectory`.

read trajectory write

`read trajectory` [*s_trj1*] `write` *s_trj2* [append] { *i_fromFrame i_toFrame* | *I_frames* }
a trajectory **editing** tool. Read ICM-trajectory `file` with the MC simulation trajectory, grab a fragment [*i_fromFrame:i_toFrame*] and append it to some other file *s_trj2*.

read object

read object [*s_objFileNameRoot*] [number= { *i_objNumber* | *I_objNumbers*}] [delete] [name= *s*]
read previously formed and saved ICM-molecular-object file. If ICM object file contains several objects, all the objects are read. If argument *i_objNumber* is specified only the specified object is read.
The names of the loaded objects from are stored in the S_out array, and the number of new objects in i_out .
Options:

- delete : temporarily sets l_confirm to no and, consequently, overwrites objects with the same name without a confirmation.
- number = i|I : reads one or several objects for a multi-object file
- name = s_ : redefines the object name

See also: build command to create an object from the sequence and copy object command to copy the existing object.
Example:

```
read object "1crn"
read object s_xpdbDir+"4tna" name="tmp" delete

build string "se glu" name="glu"
build string "se his" name="his"
write object a_1. "obb"
write object a_2. "obb" append
delete object a_*.
read object "obb" number=2
S_objNames = S_out
show a_$S_objNames[1].
```

Some properties of the current object a_ which can be extracted (most of them are also applicable to any selection):

- Box(a_) - bounding box
- Cell(a_) - crystal cell
- Charge(a_) - total charge
- Date(a_) - creation date of a pdb-file or 0
- Field(a_ iField) - one of 16 user fields.
- Field(a_ 15) - the number of missing residues
- File(a_) - the source file name or empty string
- Label(a_) - object remark
- Mass(a_) - total mass
- Nof(a_) - number of atoms
- Name(a_)[1] - object name
- Parray(a_) - chem-object (represented by smiles)
- Resolution(a_) - X-ray resolution or 9.9
- Site(a_ ..) - residue-feature information
- Smiles(a_) - smiles string
- String(a_ [number]) - string representation of the object selection
- Sstructure(a_ [compress]) - secondary structure string of all molecules
- Symgroup(a_) - symmetry group as string
- Transform(a_ ["bio" i]) - crystal or bio transformations
- Type(a_ 2) - type, like "ICM" "X-ray" ..
- Xyz(a_) - atomic coordinates

read object parray

read object parray *s_obfile* [name=*svarName*]

reads objects from .ob file into a parray . In this case the objects are not loaded into the workspace but instead are stored in an array. The array can also be added to a table, e.g.

```
read object parray "threeobj.ob"  # creates array threeobj
group table t threeobj "Obj"
```

read pdb

read pdb [all [stack]|charge|delete|header|html|sstructure] [*s_pdbFileNameRoot* [.mol/res1:res2/at1,at2,..]]
read pdb-formatted file and create a molecular object of a corresponding.

You can read all the information from the file or only the part you need:

- the whole object: read pdb "/data/pdb/2ins"
- one or several chains: read pdb "/data/pdb/2ins.a,b/" (if chain is not named, refer to it as 'm')
- chain fragment: read pdb "2ins.a/3:16"
- certain atoms: read pdb "2ins./3:17/ca,c,n" (you may use name patterns with **wildcards** too)

ICM parses a PDB file and detects problems. It may issue 72 kinds of warnings and 33 kinds or errors. To check if a certain type of error occurred use the Error (*i_errWarnCode*) function. Structures determined by NMR are usually represented by several models separated by MODEL and ENDMDL fields. By default only the first model will be read in.
Options:

- all : may be used to load all NMR models. Each model will be placed into a separate object. Object names will be automatically generated. This option is not necessary if TOOLS.pdbReadNmrModels is set to "all" .
  stack is an additional to all option for reading all models in a multimodel pdb file. It leads to an internal stack for a single object, instead of creating explicit objects for each model. Examples:

```
 read pdb "1htx" all     # is equivalent to
 read pdb "1htx" TOOLS.pdbReadNmrModels="all"
 read pdb "1htx" TOOLS.pdbReadNmrModels="all stack"  # or
 read pdb all stack "1htx"
 display stack a_ cartesian 20. loop
```

- bond : suppresses bonding of what appears to be multiresidue hetero molecules.
- charge : read partial atomic charges from the occupancy field
- delete : temporarily sets l_confirm to no and, consequently, overwrites objects with the *s_pdbFileNameRoot* name if found in ICM shell. with the same name without a confirmation.
- header : store the PDB entry header information in the object. In contrast to the html option it does not change it and stores the header info fully and as text. Also note that the header is assigned only to the first model (to save space). The header is returned by the Header(*os_*) function for multiple objects and Header(*os1_*) [1] for one object.
- header html : both options will modify the behavior of the header option. The <BR> tag will be added before the carriage return at the end of teach line.
- html : store some the PDB entry comments and header data in the object. See also Header function. The fields parsed are the following:
  - HEADER
  - COMPND
  - SOURCE
  - REVDAT
  - JRNL AUTHOR,TITL and REF
  - CRYST?
  - FORMUL
  - REMARK starting from: AUTH, TITL, REF, REFN, and also REMARK ... RESOLUTION
  - REF and REFN
  - HET
- sstructure : redefines the secondary structure by analyzing the pattern of hydrogen bonds (see assign sstructure)

Deleting alternative atoms Frequently there are alternative atoms in PDB objects. Sometimes you want to get rid of all secondary alternatives and make the 1st alternative the detault. To achieve follow this example:

```
read pdb "1hyt"
set comment a_//Aa,A1 " "  # clear the alter-symbol of the main alternative
delete a_//A               # delete atoms with non-space alter-symbol
write pdb "clean"          # this object does not have alternatives
```

Error detection.
ICM detects chain missing residues according to the differences between SEQRES sequence and the residues with coordinates and returns the total **number of missing residues** in the `i_out` system variable. This number can also be returned by the `Field( a_ 15 )` function. E.g.

```
 read pdb "1amo.a/"
 make sequence a_1.1  # sequence 1amo_1_a extracted
 if(i_out>1) then
   read pdb sequence "1amo"  # sequence 1amo_a read
   a=Align(1amo_a 1amo_1_a)
   build model 1amo_a a_1.1 a   # patch the missing fragments
 endif
```

See also: `convert` command to turn it into an ICM-molecular object and the `FILTER` preference to see how to read the compressed pdb-files directly.
The fields parsed by ICM.
ICM parses most of the information from the PDB database entry and allows one to manipulate with this information in the ICM-shell. The following fields are parsed:

- ATOM : all atom properties including alternative chains. To show the info: `show a_//*`. Function to extract the atom properties:
  - ◊ `Bfactor`: b-factors
  - ◊ `Charge`: charges
  - ◊ `Occupancy`: charges
  - ◊ `Name`: atom names

  You can also `select` by many different properties of atoms, residues, molecules and objects directly in the selection expression or via the `Select` function.
- HETATM : all properties including alternative chains (to clear the flag, use `set comment` *as* " ")
- EXPDTA : assigned as the ICM-object type. ICM function: `Type(` *os_*).
- REMARK 2: resolution is extracted. ICM function `Resolution(` *a_*).
- REMARK 4: is shown as info upon reading.
- REMARK 800: description of `SITE`s is extracted. Can be viewed by `show site`. You can select these `sites` by a_/F" *siteID"*
- COMPND : assigned to the object comment field. Editable and reassignable with the `set comment`. The comment is returned by the ICM function `Namex` . You may directly select with the a_"searchString". expression.
- SSBOND:
- DBREF: database reference information shown upon reading
- SITE : `sites` can be shown with the `show site`, can be selected with the `a_/F` expression.
- HELIX : returned with the ICM `Sstructure` function.
- SHEET : returned with the ICM `Sstructure` function.
- SEQRES: this sequence can differ from the sequences extracted from the ATOM records. It is read with the `read pdb sequence` command and becomes an ICM-shell `sequence`
- SCALE,TVECT,MTRIX: read but not used, the CRYST1 and ORIGX information is used instead.
- CRYST1,ORIGX : the `transformation vector` is returned by the ICM function `Symgroup` and can be applied with the `transform` command.
- Date of creation of the file (part of the HEADER record) can be returned by the `Date( a_ )` function.
- The number of residues missing in the density but present in the SEQRES record (`i_out` or `Field(a_`,15)).

Treatment of water molecules. Water molecules become molecules named sequentially `w1`,w2,w3... Their original numbers which are stored in the residue field become their 'residue' numbers, e.g. to select water molecule number 225 and 312, do not use the `w..` names of water molecues, but use the `a_w*/225`,312 selection instead.
Option `charge` tells the program to load atomic charge from the occupancy field and reset occupancies to 1., and atomic radii from the B-factor field.
Option `sstructure` tells the program to automatically assign the secondary structure if it is not provided in the PDB entry.
The file will be first searched in the local directory. Extensions *.pdb and *.brk will be tried unless explicitly specified. If not found the `s_pdbDir` directory or directories will be looked up according to the `pdbDirStyle` preference. This preference allows file names like pdb1abc.ent recognized by the `read pdb "1abc"` command.
Examples:

```
read pdb "1crn"          # 1crn.brk should be either in the local
                         # directory or in s_pdbDir one

read pdb "2ins.a/"       # load only chain 'a'

read pdb "2ins.a//ca,c,n" # load only the backbone of chain 'a'

read pdb "1crn./4:17"    # load only 4:17 fragment from 1crn.brk
```

See also: `read binary pdb`

read pdb sequence

`read pdb sequence` [`resolution`] [*s_pdbFileNameRoot*]
quickly extract only amino-acid sequence from SEQRES records of a pdb-formatted `file` **without** actually loading molecules. This option does not work with `pdbDirStyle` = "PDB ftp-site" or "PDB web-site" .
It is important to understand that sometimes sequence from the SEQRES records **does not match** the sequence extracted from the ATOM records, because some residues in flexible loops and ends are invisible. Option `resolution` appends X-ray resolution to the sequence name (like 9lyz_a19, 19 stands for 1.9 resolution). 'No' is appended for NMR and theoretical structures. It can be used later by the `group sequence unique` command to compile the `representative list` of PDB chains.
PDB is famous for having numerous errors which are never fixed. In SEQRES sometimes the stated number of amino-acids in SEQRES does **not** correspond to the actual number of amino-acids (e.g. 1cty, 1ctz, 1ctz, 2tmn, 1ycc, 2ycc) .
The sequences will be called according to the pdb code and the chain name. In case of one chain without a name, ICM assigns name `"m"` . e.g. 1est_m , 2ins_a , 2ins_b.
Records are converted to lower case. In rare cases, such as `1fnt`, in which there are both upper and lowercase chain names, the lowercase names become uppercase, e.g. `1fnt_a` for the first chain and `1fnt_A` for the 33-rd chain.
Chains with numerical chain identifiers are automatically converted to literal chain IDs in the same way as the `read pdb` procedure does that. Chain `0` becomes `a` , chain `1` becomes `b` , etc.
An example script to detect problems with pdb sequences (you can build the list with the `makeIndexPdb` and `mkUniqPdbSeqs` macros)

```
read sarray s_pdbDir + "pdb.li" name="a"
l_info = no
errorAction = "none"     # otherwise breaks at pdb1aa5.ent
for i=1,Nof(a)
  read pdb sequence s_pdbDir + a[i]
  delete sequence
endfor
Error> no SEQRES records in file  /data/pdb/af/pdb0af1.noc.Z
Error> no SEQRES records in file  /data/pdb/ao/pdb1ao2.ent.Z
Error> no SEQRES records in file  /data/pdb/ao/pdb1ao4.ent.Z
Warning> Sequence of chain  "pdb1ati_c"  starts with 'UNK' and is unknown
Warning> Sequence of chain  "pdb1ati_d"  starts with 'UNK' and is unknown
..
```

The number of residues which are present in the SEQRES record, but are missing from the ATOM records is returned by the `i_out` variable, or `Field( a_ 15 )` function.

read profile

`read profile` [*s_prfFileNameRoot*] [`name=` *s_prfName*]
read ICM-sequence profile from a `file` and create an ICM-shell variable of `profile` type.

read prosite

`read prosite` [*s_prositeFileName*]
read all the patterns from the prosite database (Amos Bairoch, University of Geneva, Switzerland) and create two string arrays: `prositeNames`, and `prositePatterns`, containing names and patterns, respectively. The search may be performed by the `find prosite` command. Check also the `find prosite` command.
Examples:

```
read sequence "zincFing.seq"  # load sequences
```

```
  find prosite           # search all 1374 patterns
                         # through the sequence
```

See also: `s_prositeDat` .

read rarray

`read rarray` [*s_rarrayFileNameRoot*] [`name=` *s_RName*]
read `real array` from a file. File `format` is free.

read blob

`read blob` [*s_fileName_or_URL*] [`name=`*s_blobVar*]

read any data from ~s_fileName_or_URL or standard input into `blob` shell variable.

See also: `blob Blob`

read sarray

`read sarray` [`connect`] [`comment`] [`underline=`*i*] [`number=`*i*] [*s_fileName*] [`name=` *s_varName*]
read any text from a `sar-file` as a bunch of strings separated by carriage returns. Create an ICM-shell variable of `sarray` type.

Options:

- ♦ `comment` : skips comment lines starting from hash (#).
- ♦ `connect` : will connect/merge several consecutive lines if a continuation symbol (backslash) is found at the end of the line.
- ♦ `underline=` *iFirstLine* : skips lines before this number
- ♦ `number=` *nLines* : read only specified number of lines

Example:

```
line1 \
 continue in line 1\
 more to line 1
line 2
```

will turn into:

```
#>S a
line1    continue in line 1   more to line 1
line 2
```

* `underline=` *N* : reads only under specified line (skips *N* first lines).

reading large data amounts by chunk

`read sarray` [`limit=`*n_records*] [`keep`] [`separator=`*s_sep*] [*s_fileName*] [`name =` *s_varName*]

reads up to *n_records. s_sep* is used as a record separator. Sets `l_out` to *yes* if the end of the file is reached.

keep option keeps file open for the next read chunk. Without *keep* the command will always read from the beginning of the file.

Example: (read uniprot file)

```
while (yes)
 read sarray keep separator = "//\n" limit=100 "/data/uniprot/uniprot_sprot.dat" name="
 if (l_out) break
 for i=1,Nof(s)
        Match(  uniprot_sprot[i], "ID\\s+(\\S*?)" 1 )
 endfor
```

```
endwhile
```

`read table mol` command is also supports *limit* and *keep* options

Example:

```
while (yes)
 read table mol keep limit=1024 "large.sdf" name="t"
 if (l_out) break
 # process 't'
 print Nof(t)
endwhile
```

read sequence

`read sequence` [`group` [= *s_groupName*]] [`fasta` [`auto`|`selection`..] | `pir`| `gcg`| `msf`] [*s_seqFile*]
read amino-acid or DNA sequence from a variety of `sequence file formats` and create an ICM-shell variable of `sequence` type. The GeneBank format is recognized automatically. Option `group` with optional *s_groupName* creates a sequence group on the fly.

Option `auto` (with `fasta` or `pir)` will create an alignment if all sequences (with dashes) in a multiple fasta file have the same lengths.

Option `selection` (with `fasta` or `pir)` will create a selection if the sequences read.

`read sequence swiss` [`field=` *S*] [`group` [= *s_groupName*]] [*s_seqFile*]

`read sequence swiss web` *s_swissProtName*

Example:

```
read sequence swiss web "1433B_HUMAN"
show site 1433B_HUMAN
```

Note, that if you want to ignore some types of the swissprot `FT` feature table, e.g. HELIX, or COIL, see `swissFields`)
See also: `swissFields`

read sequence database

`read sequence` *T_indexSubset*
read amino-acid or DNA sequence from an `index`ed sequence database. *T_indexSubset* contains the selected entries which can be defined by a `table expression` (e.g. SWISS.ID=="^IL2_*"). The names of the sequences extracted from the database to the ICM memory are stored in the `S_out` system `string array`. `i_out` contains the number of the sequences loaded. These variables are used in automated scripts for bioinformatics (see `searchSeqDb` or `searchPatternDb)` macros.
Examples:

```
 read index s_inxDir+"/SWISS" # load the Swissprot index
 read sequence SWISS[1:20]        # first 20 entries
 show S_out[1], $S_out[1]         # show the 1st name and the sequence
#
 read sequence SWISS.ID=="^IL2_*" & SWISS.ID!="*_MOUSE"
 S_seqNames = S_out
 for i=1,Nof(S_seqNames)
   seqName = S_seqNames[i]
   show seqName, Nof(String($seqName),"[KR]") # stat. of positive charge
 endfor
```

read stack

`read stack` [`append`] [*s_stackFileNameRoot*]
read `stack of conformations` from a `cnf-file`. This command resets the `energy terms` as they were saved in the cnf-file. The terms string is returned in the `s_out` variable.

Both full stacks saved with the `write stack simple` command and compressed stack files (the default) will be recognized. Note that ICM versions before 3.022 could not read or write the compressed format.

read string

`read string` [*s_textFile*] [`name=` *s_sName*]
read any text from either standard input or a *s_textFile.* Place the result into the `s_out` string. Reading string from standard input can be used to get URL-encoded stream generated by the HTML-form. The read string command can also read from `ftp/http`.
See also: `read unix` command which allows one to read in ICM the output of any unix command.
Examples:

```
cat someFile | icm -s -e "read string;Tolower(s_out)"
```

more:

```
#Put these lines into _tmp file. See how to precess the HTML-form output.
 read string     # e.g.: <b>echo "aaa=bbb&ccc=ddd" | icm _tmp</b>
 a=Table(s_out)  # split the input string into two string arrays
                 # a.name and a.value and form table 'a'
 show a          # equivalent to <b>show column a.name a.value </b>
 quit
#
 read string "ftp://ftp.pdb.bnl.gov/index/compound.idx" name="pdbList"
```

In the last example the file will be downloaded from the PDB site and dumped into the pdbList string variable.

read table in ICM or CSV/TSV format

ICM-formatted tables `read table` [`database`] [`name=` *s_tableName*] [*s_tableFileName*] [`split=` *s_fieldDelemeter*]
reads internal ICM text format for tables. It has fields for the table headers. The table name is saved to the `s_out` variable. ICM needs two lines with the table name and the field names in the following format: (an example):

```
#>T atm
#> name code weight
hydrogen 1    1.008
....
```

`s_fieldDelimiter` is NOT used in the ICM table reader. If you want to change the default field delimiter use the `split=` *s_fieldDelimiter* argument. To skip multiple occurrences of a delimiter symbol, repeat it two times, e.g. `split=` " \t\t" (the same trick is used in the `s_fieldDelimiter` variable for the `Field` function)
CSV or TSV formatted tables

`read table separator=`[","|"\t"|":"..] *s_csvtableFileName* [`delete`][`header`][`simple`] [`name=`*s_tableName*] [`comment=` *s*]
reads tables in portable **csv** (comma-separated-value) or **tsv** (tab-separated-value) formats. Options:

- `comment [=]` : skips lines beginning with the symbol (pound sign # is the default)
- `delete` : deletes table with the same name
- `group` : group multiple columns of real type into one column of vectors (same with integer columns)
- `header` : interprets the first line as the names of the columns.
- `number` : treat **empty fields** in numeric columns as `ND` (the default action is to keep those columns as string arrays)
- `simple` : *quotes* are not treated as regular characters

Flanking blanks for each field are trimmed. For example to read the following table from iq.csv file:

```
name,IQ
  Max, 150
```

```
 Jack, 150
Peter, 130
```

type:

```
read table separator="," header "iq.csv"  # or
read table separator="," header "iq.csv" name="t" # to rename the table
```

Normally the csv/tsv format does not allow any line comments. ICM supports an extended format in which some lines can bee commented out by a comment string in the beginning of the line, e.g.

```
> cat iq.csv
# this is a list of IQs
name,IQ
Max,  150
Jack, 150
Peter,  130
> icm
icm/> read table separator="," header "iq.csv" comment="#"
```

See also: `table`, `icm.tab` file, `add column` .
Examples:

```
 read table s_icmhome+"atm" name="ATOMS"  # atm.tab file by default
 sort ATOMS.weight   # sort according to the weight array
```

Reading SMILES file into a table

`read smiles` [`header`] *s_filename* [`name=`*s_newTableName*]

reads the *s_filename* file in `smiles` format into a table. Table name is derived from the file name if *s_newTableName* is not specified explicitly.

With the `header` option specified, the first line in the file is used for table column names. Example of the space-separated smiles file with header:

```
chem prop
CCC  1.0
CCCC 2.0
```

`read smiles header` of this file will create a table with two columns `chem` and `prop`

Reading an html table into an ICM table

`read table html` *s_htmlFile|or URL* [`name=`*s_newTableName*] [`all`] [`header=`[`yes`|`no`]] [`simple`]

this command will read a file containing one or several html tables, then will select the largest table (by the number of rows) and read it into an ICM table.

Options:

- ♦ `all` read all tables rather than the largest one;
- ♦ `header` interpret the first row as column names even in cases when the column name row is incorrectly marked with the `TR` tags instead of the correct `TH` tags;
- ♦ `header=no` will do the opposite: force the reader to read the first row as a table row;
- ♦ `simple` remove all HTML tags from the cell values.

Reading an mmcif-file into an ICM chemical table.

`read table mmcif` *s_mmcifFile*

reads a pdb mmcif file with multiple small molecules (not for the whole pdb) into a chemical table. The short description of `mmcif` format is given below. The full description is provided by pdb .

This command will create a chemical table and all general properties will be converted into columns.

Reading an MOL2-file into an ICM table.

`read table mol2` *s_mol2FileName*

reads an `mol2` file into an ICM table which can be visualized as a chemical spreadsheet.

Reading an sdf-file into an ICM table.

`read table mol` [`exact` | `unique`] [`simple`|`simple=`*S_cols*] *s_sdfFileName* [`index`] [`limit=`*i* [`keep`]]

`read table mol` [`exact` | `unique`] *T_sdfFileIndexExpression* [`index`]
reads an `sdf` file into an ICM table which can be visualized as a chemical spreadsheet. It either reads all entries directly from the file, or read the entries selected by the index expression (e.g. `chemvendor[{1,15,53}]`). The the latter case the `index file` needs to be read in first. In contrast to the `read mol` command, the `read table mol` command creates only a table and does not create explicit ICM molecular objects Consequently it can read over hundred thousand mol-records into a table without overwhelming ICM.

The table name is saved to the `s_out` variable.

The property fields of the sdf file, e.g.

```
> <logP>
 2.3
> <logD>
 1.8
```

are converted automatically into table columns with appropriate type. The mol-file core which describes atoms and bonds is automatically displayed as a chemical structure by ICM. By default the empty property fields are interpreted as having 0. value, if all non-empty fields are numerical. Options:

- `simple` : keeps all columns as string arrays. Optionally a list of column names can be provided: simple = {"col1","col2"}. In this case only listed columns will be kept as string arrays
- `exact` : keeps columns containing numbers and empty fields as string arrays instead of trying to guess the numerical default value for those columns.
- `index` : creates an extra column named `IX` in which the compound order number in the file is stored. If property `IX` already exists in the file, its values will be overwritten.
- `keep` : preserves the file pointer and allows one to read the NEXT *frame* (or group or rows) with the next `read table mol` command. See also: `l_out` to indicate if the next read is possible.
- `limit=` *n* : determines the size of the chunk to read at a time.
- `unique` : standardizes the property field names. For example, "Molecular Weight", "MWeight", "Mol_weight" will be translated into "mw". This option may be helpful if you want to merge two sdf files.

Example:

```
%icm -g
read table mol unique "sigma.sdf"

write index mol "sigma.sdf" "sigma.inx"
read index "sigma.inx"
read table mol sigma[{1,15,26}]
```

Reading chunks from t.sdf and spitting out chunks of the itb stream:

```
  while yes
    read table mol "t.sdf" keep name="t" limit=1024
    if (l_out) break
    write binary frame t
```

```
 endwhile
```

See also:

- ♦ `write table mol` command,
- ♦ `Nof (` t.mol s_smartExpression)
- ♦ `read smiles` [header] : reads space separated smiles with properties
- ♦ `read table mol2`

read table into arrays

`read column` [`separator=` *s_Separator*] [`group` [`name=` *s_tableName*]] *s_fileName*
read a multicolumn table with strings and numbers and create appropriate arrays. If you add a ruler starting from #> and looking like this

```
#>-name1---name2------name3---------name4---
```

the arrays will be created with specified names. If ruler is missing, default names (I1, I2 ..., R1, R2,..,S1, S2, .. for iarrays, rarrays and sarrays, respectively) will be created. You may control field formation by `s_fieldDelimiter` variable or by adding **separator=** *s_Separator* explicitly. The list of array names will be stored in `s_out` so you can always say

```
 read column "res"
 show column $s_out

# note that a triple quote permits multiline entry
read column group name='t' input="""
a 1 2.2
b 2 3.2
"""
show t
```

Another way to read a table into ICM arrays is to read it as table with the `read table` command and split the table afterwards.
Reading comma-separated-value or tab-separated-value formats
While the best way to read a `csv` file is to use the `read table separator=","` command, you can use the `read column group` command as well. To read a table in comma-separated-value (*csv*) or tab-separated-value (*.tsv*) format redefine the `s_fieldDelimiter` value (or use the `separator=","` option), and use the `read column group` command.

```
read column group name="t" "t.csv" separator=","
write t separator=","
```

See also: `write column`, `read table` , split table , `show column`, `icm.col`.

Reading internal variables from a file

`read variable` [*s_varFileNameRoot*]
read ICM-molecular object variable values (torsion angles, phase angles, bond angles, bond lengths) from a `var-file`. `vs_out` selection will contain a selection of variables which have been modified by the command. Variables are assigned according to the residue number and the variable name. If residue name is different (i.e. you want to assign phi,psi of an alanine 15 to glycine 15), the program sends a warning. If more than one molecule is present in the current object, matching of molecule names is required. See also `set vs_` command.

Reading and setting a vew from a file of view parameters

`read view` [*s_viewRarrayName*]
read `rarray` of 37 display parameters for window size, scale, view matrices, etc. and set them.
See also: `set view`, `View` () function
Examples:

```
 build string "se ala"
 display
 write View( ) "a"
```

```
# rotate the image
 read view "a" # restore view
```

Reading vrestraints from a file

read vrestraint [*s_rsFileNameRoot*]
read variable restraints (often referred to as rs) from a *.rs file. Do not forget to read vrestraint types first. Option only tells the program to delete previous variable restraints.

Reading vrestraint types from a file

read vrestraint type [*s_rstFileNameRoot*]
read variable restraint types from a *.rst file. Option only tells the program to delete previous variable restraint types.

Reading XML from a file

read XML formated document into a hash(formely collection) object.

read xml { s_fileName | s_url | input=s_xmlBuffer } [name=s_name]

Example:

```
read xml input="<a>1</a>" name="x"
#

read xml name='x' "http://www.drugbank.ca/system/downloads/current/drugbank.xml.zip"
show name x
```

See also: array , xml drugbank example

Reading JSON from a file

JSON (an acronym for JavaScript Object Notation pronounced) is a lightweight text-based open standard designed for human-readable data interchange. Read more here.

read JSON formated document into collection object.

read json { s_fileName | s_url | input=s_jsonBuffer } [name=s_name]

Example:

```
read json input='{ "a":"b", "c":[1,2,3]}' name="x"
```

rename

rename *oldName* { *s_newName* | *u_newName* }

rename atom *as1 s_newElement*\n\

rename *as1*|*rs1*|*ms1*|*os1 s_newName* # rename *os* full -for description\n\

rename image *P_imageArray i_index s_name*\n\

rename page *P_pageArray i_index s_name* # see below, for icmdb\n\

rename sequence resolution # from 1abc_a to 1abc_a21 for sequences linked to a_A. used in group unique..\n\

rename anything to anything else. More specifically you can rename commands, ICM-shell variables, objects, molecules, residues and atoms. Renaming commands is possible, but then you must not forget to change them in all the standard ICM-scripts. Using `aliase`s instead allows you to use both the original and the translation, however it slows down the ICM-shell interpretation. Be careful with a new name to avoid name conflict.

rename object

`rename` { *os* [`full`] | *ms* | *rs* | *as* } *s_newName*
change selected names. To change the long name of the object (it can contain space in contrast to a regular object name), use the `full` option.

If you rename **multiple** molecules and provide a *s_neweNameRoot* (say, `"a")` at attempt will be made to name them like this: "a","a1","a2",... .
Examples:

```
rename old   mature          # for elderly
rename sequence[1] ins       # rename the first sequence
rename a_mol1/3/ca "ca1"     # rename an atom
rename a_mol1/3    "alam"    # rename a residue
rename a_mol1      "kuku"    # rename a molecule
rename a_H         "h"       # all heteroatoms h,h1,h2,h3,..
rename a_1.        "dna"     # rename an object
       # rename the full name of the object
rename a_1. full "hydroxanthine phosphoribosyl trnasferase"
list a_1.
```

Groups of atoms also can be renamed from a chemical (2D) template with the `set bond topology` *as chem_source* `label` command.

rename molcart table

`rename molcart table` *s_oldTableName s_newTanbeName* [*connection_options*]

renames Molcart table including all index tables. Database connection may be specified by *`connection_options`* The table name *s_oldTableName* may be prefixed by the database name, else current database is assumed.

See also: `molcart`, `list molcart`

rename/move file

`rename system` *s_fileNameFrom s_fileNameTo*

Renames or moves a file. If target file exists it will be overwritten.

Example:

```
rename system "/tmp/aaa" "/tmp/bbb"
```

See also: `sys` , `delete file` `delete directory`

restore preference

`restore preference` [`all` | *prefname1 prefname2* ...]

restores values of the named (or all) GUI preferences to the default values.

return

`return` [`error`] [*s_message*]
return from a macro before `endmacro` usually under specific conditions. Similar to `exit` command returning from a file to interactive mode. Option `error` will set the error flag which can later (outside the macro) be checked with the `Error( )` function. The message *s_message* will be stored in the `s_out` string shell variable.
Examples:

```
 macro aa
    if(Nof(sequence)==0) return   # a silent return
    .....
  endmacro

  macro aaa as_1
    if(Nof(as_1)==0) return error " aaa> Nothing to do"
    show as_1     # a pretty silly macro
  endmacro

  macro bbb
    if(Nof(object)==0) return error " Error_in_bbb> No objects in the system"
    ....
  endmacro
  bbb             # call this macro
  if(Error) print "something went wrong with macro bbb"
```

rotate

the main `rotate` command. Subtypes of this command include `rotate object`, `rotate view`, `rotate grob`. `interruptible background rotations and rocking movements`.

Also, to perform a fixed number of interruptible rotations or rocking movements, use this:

```
GRAPHICS.rocking=5  # for X-rotation. see other types of rocking/rotation
GRAPHICS.rockingSpeed=3.
display rotate 2  # perform two full cycles and stop
```

rotate object

`rotate` [*os* | *ms* | *g_grob*] *M_rotation*
rotate an object (*os_*), one/several molecules (*ms_*) or *g_grob* with the specified rotation matrix.
Examples:

```
 rotate a_1. Rot({0. 0. 1.},30.)  # rotate by 30 degrees
                                  # about Z-axis
```

See also: `interruptible background rotations and rocking movements`.

rotate grob

`rotate` *g_grobName M_rotation*
rotate a graphics object.
Examples:

```
 read grob "oblate"
 display g_oblate magenta
 rotate g_oblate Rot({0. 0. 1.},30.)
```

See also: `interruptible background rotations and rocking movements`.

rotate view

`rotate view` *M_rotation*
rotate view in the graphics window with respect to the screen axis X (horizontal), Y (vertical) and Z (perpendicular to the screen). This command is great for creating movies or demos when the graphics should be manipulated from a script.
Example:

```
 build "alpha"
 read trajectory "alpha"
 display a_//ca,c,n
 for i=1,100
   load frame i
   rotate view Rot({0. 1. 0.} , -1.) # rotate around Y by -1 deg.
 endfor
```

See also:

- the `View` () function
- the `set view` command.
- `interruptible background rotations and rocking movements.`

select

Many ICM shell objects may have some parts *selected* in order to perform various actions on selected parts. The **select** family of commands allows one to create/modify/remove selections of ICM objects of subitems in ICM objects.

The main select commands:

`select` [`off`] [`alignment` | `grob`|`iarray`|`map`| `matrix`| `profile`| `rarray`| `sarray`| `sequence`| `table`..] [*s_namePattern*]

(un)selects ICM objects with certain name pattern (or all), e.g. `select grob "g_pocket*"`

`select chemical` ... select patterns in `2D chemicals` of a chemical table

`select` *alignment as*

`select` [*dist*|`distance`|`hbond`] *as*

`select` *treeArray* `center`

See also:

- selections in `molecular objects` are treated differently.
- `Select` function
- `select by center of mass`

Align/color/select chemical by pattern or other properties

This command can be used for:

Align 2D chemical by pattern (with *rotate* option)

`select chemical` *chemarray* {*s_smarts*|*chempattern*} `rotate` [`index=`*i*|*I*]

Select matched fragments(s) (with *all* and/or *append* option)

`select chemical` *chemarray* {`s_smarts`|`chem`} [`all`] [`append`]

Color matched fragments (with *color* option)

`select chemical` *chemarray* {`s_smarts`|`chem`} `color=s_color` [`all`] [`append`]

Use atom-level predicate *s_filter* to color/select individual atoms.

`select chemical` *chemarray* `filter=`*s_filter* [`append`] [`color=`*s_color*]

s_filter should contain a logical expression which may use certain atom-level properties.

- Mass - returns atomic mass
- NofHeavyBonds - returns number of heavy neighbors
- NofHydrogens - returns number of attached hydrogens
- Name - returns atom name
- Color - returns currently assigned color
- HBA - returns 1 if atom is hydrogen bond acceptor, 0 - otherwise
- HBD - returns 1 if atom is hydrogen bond donor, 0 - otherwise
- Chiral - returns 0 for non-chiral, 1 for S, 2 for R, 3 for undefined/racemic.
- nRng - returns the number of rings atom is member of
- Code - returns atomic number
- IsOrganic - return 1 if atom is H,C,N,O,S,P,Se,F,Cl,Br,I. 0 - otherwise
- Valency - returns atom's valency

- Hyb - returns atom's hybridization state (possible values: 0,1,2,3)
- AtomNum - returns atom's order number in connectivity table or smiles.
- Aromatic - returns 1 if atom is aromatic, 0 - otherwise.

Examples:

```
add column t Chemical({ "CC1C=CC(C(NCCC(C=CC(S(NC(NC2CCCCC2)=O)(=O)=O)=C2)=C2)=O)=NC=1"
 "C#CCN1C(=NC(c2ccc(cc2)S(N2CCCCC2)(=O)=O)=O)Sc2cccc(c12)[Cl]"})
# color all hydrogen bond acceptors
select chemical t.mol "[O,S&v2,N&^2&X2,N&^1&X1,N&^3&X3]" all append color=lightblue
# the same but using atom-level expressions
select chemical t.mol filter="HBA" color=lightblue
# select all SP3 atoms
select chemical t.mol filter="Hyb==3"
# color first atom of the first molcule
select chemical t.mol filter="AtomNum==1" color=red index=1
# color all hydrogen bond donors
select chemical t.mol "[!#6;!H0]" all color=lightred
# align molecules by scaffold
select chemical t.mol "C(=C(C=CC1S(=O)(=O)[R2])[R1])C=1") rotate
# select all methyls and delete them
select chemical t.mol "[C;D1]" all
delete chemical selection t.mol
```

See also: `delete chemical selection` `find table`

select-3d-label

`select` [`edit`] *3d-label*

graphically selects the specified label. It appears as a little green cross. The labels are considered as a subclass of `graphical objects`.

Options:

- `edit` : displays the label handle allowing to drag the label

See also:

- `make 3d label`
- `set label` *label_3D*

set family of commands

to change properties/attributes of existing icm-objects.

`set area`|`atom ball`|*as*
..|`background`|`ball`|`bfactor`|`bond`|`cartesian`|`chain`|`charge`|`chiral`|`color`|`comment`|`comp_`
 `conf`|
`directory`|`drestraint`|`edit`|`error`|`factor`|`field`|`font`|`foreground`|`formal`|`format`|`grob`
 `index`|`key`|`label`|`link`|`map`|`menu`|`mmff`|`molcart`|`name`|`object`|`occupancy`|`plane`|`property`
.. |`randomize`| `reflection`|`resolution`|`selftether`|`separator`|`sequence`
`reverse`|`site`|`slide`|`sstructure`|`stack`|`stereo`|`stick`|`swiss`|`symmetry`|
`table`|`tautomer`|`term`|`tether`|`texture`|`topology`|`torsion`|`tree`|`type`|`view`
arguments

Example:

```
 set bfactor a_//c* 20.
```

set area sequence : positional factors for sequence alignment

`set area` *seq* [{ *R_factors* | *r_factor* }]
sets/resets a property array assigned to a sequence. Each amino acid can be assigned a relative solvent accessibility value for this residue in a three-dimensional model. 0. - fully buried (the highest possible factor), 1. - fully exposed. These values can also be used to influence the alignment (buried residues with accessibilities close to zero will have larger contributions). The exact dependence residue-residue score factor on this value is defined by the `accFunction`

array.
`set area` *rs* [{ *R_areas* | *r_area* }] sets/resets an array or accessible area values (or value) to the residue selection. Note that for the residue areas contain absolute values (e.g. 84., 120., etc.) while for sequences (see above) the area/weight values are relative accessibilities in the range [0.,1.]. The maximal possible accessibilities are returned by the `Area(` *rs_* `type`) function. Example:

```
read object s_icmhome+"crn.ob"
show surface area
set area a_/asn Area(a_/asn type) # reset areas to maximal values

set area a_crn.m 0.
set area a_crn.m/ Random(0. 1. Nof(a_crn.m/))
```

See also: `accFunction` , `Align` (*seq1 seq2* `area`)

set atom label or the ball radius

`set atom ball` *as R* sets custom balls to atoms. Can be returned with the `Radius(` *as*) function

`set atom label` *as S* sets custom labels to atoms. Can be returned with the `Label(` *as*) function

set atom

(re)sets atom properties, such as atom presence (`on` and `off`), coordinates,

`set` *as* `on`|`off` activate (unhide) or inactivate (hide) atoms for energy and surface calculations. The inactive atoms can be shown in graphics as shaded wire models. Example: `set a_//h* off ; Nof(a_// off) ; set a_//h* on` By default all atoms are active.

See also: Nof(on|off), `set atom ball label`

`set` *as_Natoms M_3xN* [`mute`]
`set` *as_Natoms M_3xN* [`mute`] # eg set a_W//vt1 Xyz(a_W//o)

`set` *as_tethered_inICMobj* [`tree`] # follows tethers in this case the target coordinates are simply taken from the destination positions for the tethers.

`set` *as_one_vt1_atom_inICMobj* [*R_3xyz*|*M_3x1*|*as*]

`set` *as_nonICM_X as_ICMtetheredToX* # used to update sdf/mol coordinates after optimization. this command is used to inherit the coordinate changes in a converted object with different order of heavy atoms. Imagine the following scenario:

- a mol file is read into a nonICM object,
- this object is converted with option tether (the order of heavy atoms may change)
- the converted object is optimized and the heavy atom coordinates are changed
- now there is a need to transfer the changed coordinates back to the source atoms in the nonICM object.

The latter is achieved via the above `set` command.

`set` *rs s_secondaryStruct* # `e.g. set a_/1:3 "HHH"` this command sets phi and psy angles of the selected residues in an ICM object according to the secondary structure

`set chain` *ms s_chainSymbol*

With a single atom selection, ICM sets a given atom to the center of gravity of the corresponding molecule (no arguments), given point in space (*R_3Dvector* argument) or center of gravity of selected atoms (*as_select* argument).
If multiple atoms are selected, ICM sets the specified atoms to their new XYZ positions. The XYZ matrix can be returned by the `Xyz` (as_) function.

If multiple atoms are tethered the coordinates of the tethered atoms can be set to the coordinates of the target atoms (see also `minimize tether`, `superimpose` and `minimize "tz"`.

Examples:

```
 build string "se ala his glu"
 set a_/3/ca Matrix(Mean(Xyz(a_//ca)))  # 3rd Ca to the center of mass of all Ca s
 set a_/3/ca Matrix({-3., 12., 14.5})

 set a_//vt1  # set the first virtual atom to the center of mass
 randomize a_//vt1 0.1  # randomize the vt1 position in case of singularity
```

For ICM molecular objects, in the most popular operation (set a_1//vt1) the first of the two `virtual` atoms (vt1) attached to the beginning of the selected molecule is set to the center of gravity of the same molecule. The purpose of this action is to simplify molecular rotation and translation via the first six free virtual variables. The tvt2 and tvt3 torsions and avt2 planar angle determine rotation of the whole molecule around the axes passing through the center of gravity. Useful for docking.
Examples:

```
 read object s_icmhome+"complex.ob"
 set a_1//vt1           # now it is easy to rotate the 1st mol.
                        # by changing tvt1
 set a_2//vt1           # now it is easy to rotate the second molecule
 set a_2//vt1 {1. 1. 1.} # move it to {1. 1. 1.} point
#
# Multiple molecules: let us set vt1 for all water molecules to oxygen
# to fix the first 3 variable and keep the oxygen positions unchanged
 read pdb "2ins"
 convert
 set a_w*//vt1 Xyz(a_w*//o)
 fix v_w*//?vt1
 mc v_w*
```

See also:

command	description
`set a_/* s_secondaryStructure`	to change phi,psi angles according to secondary structure,
`virtual atoms/variables`	information about virtual atoms and variables
`move`	command which goes further and actually changes the topology of the ICM-tree.
`set cartesian`	command that assigns coordinates from a template file

set background image in graphics

`set background image` *bgimage* `full` `set background image` *bgimage* `exact center` `set background image` *bgimage* `exact` [`origin=I_pos`] [`r_scaleCoeff`]

- set image as graphics background

With the `exact` option the image will be displayed in its own resolution. If `center` option is provided it will be centered, otherwise you may specify the origin of the left bottom corner with `origin` option (default {0 0}) and/or scale coefficient. Image created in this mode is drag-able and resize-able by mouse using 'drag-atom' mode.

With the `full` option the image is scaled to the maximum size when it still fits in the window Otherwise the image is scaled so that its central part fully covers the window without margins. Aspect ratio of the image is preserved in all of the above cases.

`set background image off`

- clear graphics background (remove any images assigned)

Examples:

```
 read image s_icmhome+"splash.png"
 set background image album[ Nof(album) ]
```

set bfactor

`set bfactor` *as* { *r_NewFactor* | *R_NewFactors* } `set bfactor` *rs R_NewResidueFactors*
set B-factors of selected atoms to a specified real value (or individual values). To assign individual b-factors, provide a real array with b-factors for each atom. To assign the individual b-factors at the residue level, provide matching residue selection and *R_NewResidueFactors* array.
Examples:

```
build string "ala his trp"  # also includes N- and C- terminal groups
set bfactor a_//* 20.
set bfactor a_//ca {20.,10.,30.}   # individual atomic factors

set bfactor a_/2:18/ca,c,n 10.

set bfactor a_/* {10.,20.,30.} # individual residue factors
```

set bond type

`set bond type` *as_class1* [*as_class2*] { *i_type* }
set the bond chemical type (0 - undefined, 1 single, 2 double, 3 triple, 4 aromatic,9 quadruple,10 amid).
`set bond auto` *ms*
with the `auto` option the command automatically reassigns patterns of single and double bonds. It performs the following operations:

- ♦ identify aromatic rings in object *os_* from patterns of single and double bonds. Use preference `wireStyle = "chemistry"` (Ctrl-L) to see the bond types. This is done automatically upon reading of objects, mol and mol2 files if logical `l_readMolArom` is set to yes.
- ♦ for ICM objects, set ICM bond variable types according to bond chemical type, atom types and distance between them

Example:

```
read pdb "1crn"
display
wireStyle="chemistry"
set bond type a_//c a_//o 2 # double        # standard bonds in a/acids
set bond type a_/phe,tyr,trp/[cn][gdez]* | a_/arg/cz*,nh* 4 # aromatic
set bond type a_/as?/cg*,od,od1 | a_/gl?/cd*,oe,oe1 2
build hydrogen a_/A
```

See also: `set bond topology`

Transfer chemical structure, formal charges and bonds (or atom names), from smiles or a chemical.

`set bond topology` *as_* [*smiles*|*chem1* [`label`]]

The bond orders and formal charges for a molecule in `object` can be modified according to the smiles string or a `chemical` if they match topologically (i.e. without consideration of bond orders and formal charges). Arguments:

- ♦ *as_* atom selection in an ICM or non ICM object, it can also be a selection of different level *rs_*, *ms_* or *os_* .
- ♦ *smiles* string with the new bond orders and formal charges, e.g. `"C1[N+]CCCC1"` for a charged piperedine.
- ♦ *chem1* a chemical `parray` with one `chemical` in it, it can be read from a mol or sdf file, e.g. read mol table "myNewChemStruct.sdf"
- ♦ `label` : option to transfer atom names from a 2D chemical along with bonds and charges. The atom names can be set in 2D editor with right-click and choosing "Edit Atom Label" item. They can be viewed by choosing `View/Show full atom names` in the `View` menu. The atom names are stored in the .sdf format as `M ZZC` records (e.g. `M ZZC 3 cg`).

The command works as follows:

- ♦ a *substructure* match without formal charges and bond orders is performed in all molecules and atoms selected (both ICM and non-ICM objects can be treated, but the

hydrogens are adjusted only in non-ICM ones)
- ♦ only the **1st** match is considered in case of multiple matches of a *smiles* string or a *chemical.* The command was meant to fix the whole molecule
- ♦ the bond orders and formal charges from the first match are transferred to the selection.

To apply thie command to an ICM object follow these steps:
- ♦ `strip` *os_* reduce the object to a non-ICM type
- ♦ `set bond topology` *ms_ s_newSmiles* # will fix hydrogens in the changed areas
- ♦ use `convert2Dto3D` or 3Dto3D macros if you want to change geometry or do it in the Ligand Editor.

`set bond topology` *ms_hetero*|*as_hetero* `auto`

guesses bond orders from coordinates (hybridization and angles) but only for molecules of non-ICM type marked as **HETATM** (type 'H')

set cartesian : imposing ring templates

`set cartesian` *os* [*X_3D_chem_templates*]

By default this command is trying to find chemical matches of the selected object with a set of 3D molecules in a `template_3D.sdf` file ($ICMHOME directory) and sets coordinates to the template if a match is found. The file can be modified, or one can use your own external set of templates as the *X_3D_chem_templates* array . Example:

```
 read mol s_icmhome+"template_3D.sdf" 1
 set cartesian a_
```

This command is used in the `convert2Dto3D` macro.

set chain symbol

`set chain` *ms_molecules chainSymbol*

sets the chain character to the selected molecules. Only the first character of the string is used as the chain identifier. If the chain character is not set is kept as the space symbol (' ') but is shown and can be selected as underscore (_) .

Example:

```
read object s_icmhome+"complex.ob"
set chain a_* " "   # clean up
show a_C_  # all molecules have blank chain character
set chain a_2 "A"
set chain a_1 "B"
show a_CAB
```

set charge

`set charge` *as_select* { *r_NewCharge* | `add` *r_Increment* }
sets or increments partial electric charges of selected atoms to or by specified `real` value, respectively.
`set charge` *as_select* { *R_NewChargeArray* | `add` *R_ArrayOfIncrements* }
sets or increments partial electric charges of selected atoms to or by a specified `real array`. The array assignment is useful for saving and restoring the charges.
Examples:

```
 set charge a_//* 0.

 set charge a_/lys/nz | a_/arg/cz  1.0

 set charge a_/asp/od* | a_/glu/oe*  -0.5

 oldCrg=Charge(a_//*)
 set charge a_//* 0.0
 set charge a_/asp/od* | a_/glu/oe*  add -0.5
# do something with these simplified charges
 set charge a_//* oldCrg
```

See also: `set charge formal`, `set charge mmff` .

set charge formal

`set charge formal` *as_select r_NewFormalCharge*
sets formal partial electric charges of selected atoms to or by a specified `real` value. The charge will be rounded to the nearest value proportional to 1/12th. The following values are common: +-N, +-N/2., +- N/3., +-N/4., +-N/6. Note that the formal charge can not be arbitrarily changed without appropriate changes in the surrounding `bond types`. The formal charge will be considered by the `Smiles` function.
Example:

```
 read object s_icmhome+"crn.ob"
 set charge formal a_//n -0.333   # a formal charge of -1/3.
```

See also: `set charge formal auto`, `set charge`, `set charge mmff` .

set charge formal auto

assigns formal charges according to pKa base and acids model.

`set charge formal auto` *X_chem_array|ms_sel r_pH*(`7.0`)

Example:

read table mol "t.sdf" name="t" set charge formal auto t.mol 7.0 # charge at pH=7

Note: this command support `nProc` option for parallelization.

displaying pKa values for chemicals:

```
add column t Chemical({"CCCCN","CCCNCCC","C(=O)O","CC(=O)O","CCC(=O)O"}) # we need a ch
# here is the action on table t
add column t Predict( t.mol "MolpKaBase" ) name="pkab"
add column t Predict( t.mol "MolpKaAcid" ) name="pkaa"
set label t.mol t.pkab window= {0.,14.}
set label t.mol t.pkaa window= {0.,14.}
set format t.mol comment = "only the lowest number is significant"
```

set charge mmff

`set charge mmff` *as_select*
set atomic charges according to the rules described in a series of `publications` on the `Merck Molecular Force Field` abbreviated as MMFF94 or just MMFF.
This command requires the mmff atom types (see the `set type mmff` command). Do not be surprised that the methyl groups have zero partial charges. That is how they are defined in the MMFF algorithm. This command is automatically execute if you specify option `charge` in the `set type mmff` command.
Example:

```
 read object s_icmhome+"crn.ob"
 set type mmff       # mmff atom types
 show atom type mmff
 set charge mmff     # charges
#
 read mol s_icmhome+"ex_mol.mol"
 for i=2,Nof( object )
   set object a_$i.
   display
   build hydrogen
   convert
   set charge mmff
   display ball
   color a_//* Charge(a_//*)//{-1., 1.} ball
 endfor
```

See also: `set charge`, `set charge mmff` .

set chiral

`set chiral` *as* [`0|1|2|3`] set a chiral flag for the selected atoms. The meaning of the flag:

- ♦ 0 chirality is not set
- ♦ 1 R-chirality
- ♦ 2 S-chirality
- ♦ 3 a racemic mixture of two chiral isomers

If no explicit integer flag is specified the program will automatically assign the flag from the local geometry and topology.

set color directly and without graphics

`set color` *atom_representation_or_label as color*

`set color ribbon|base|{residue label}` *rs color*

Allowed atom representations:

- ♦ `wire`
- ♦ `stick`
- ♦ `ball`
- ♦ `xstick`
- ♦ `cpk`
- ♦ `skin`
- ♦ `surface`
- ♦ `site`
- ♦ `atom label`
- ♦ `variable label`

The `set color` command is equivalent in action to the `color full` command (e.g. color a_*. full alignment). Option `full` allows one to set colors regardless of the display status.

`set color` *ali_lignment* [{*i_color_Schema_Num*|*s_color_SchemaName*}]

Sets alignment coloring schema. If no schema number is provided then default will be set. To modify existing color schemes or introduces new ones, modify the content of the `CONSENSUSCOLOR` file .

Example:

```
set color alig "icm-combo"
set color alig              # default 'consensus-strength' will be set
```

s_color_SchemaName can also be a name of the field set in `set field` command

Example:

```
read sequence s_icmhome + "seqs"
group sequence a
align a
set field a 1 "field1" Random(0., 1., Length(a ) )
set color a "field1"
```

See also: `color`

set comment

`set comment` [`append`] *os_Object s_comment*

`set comment` *ms*|*rs s_comment*
set a text comment string (or a long name) to object, molecule(s) or residue(s). This annotation is preserved in the `read object` and `write object` commands.
Examples:

```
 read object s_icmhome+"crn.ob"
 set comment append a_ "\n The template for modeling\n Energy minimized\n"

 build smiles "CCO"
 set comment a_1 "ethanol"
```

`set comment conf` [*os*] *s_comment i_conf*

sets a comment string to the stack's conformation.

Example:

```
build string "ASD"
store conf a_
set comment conf a_ 1 "initial conf"
```

See also: `Name conf` `store conf`

See also: `set comment` *s_alterSymbol as* , `Namex` function

set a flag of an alternative atom position

`set comment` *s_charAlterSymbol as_alterAtoms*
set alternative status to the selected atoms (e.g. `set comment a_//Aa " "` ,to clear the alternative flag). The alternative flag can be read from a `pdb` file. This flag marks alternative geometrical positions of atoms which are described in the previous ATOM records. For example, the same side-chain or a water molecule can occupy several positions. The symbol of alternative position (usually 'a','b' or 'c' character, since ICM converts the strings to low case) precedes the residue name field. The alternative positions can also be selected with the `a_//A` *alterChar* selection.
Example:

```
 read pdb "1cbn" # has alternative positions
 show a_//Ab     # show alternative pos. 'b'
 set comment a_//Aa "x" # rename 'a' positions to 'x'
#
# example in which we delete all secondary alternatives and
# clear the alternative-flag from the main alternative
#
 read pdb "1hyt"
 set comment a_//Aa " "  # cleared the main alternative
 delete a_//A            # delete atoms with any alter-symbols, eg b,c,2,3 etc.
```

---

set comment to a sequence

`set comment` [`append`] *seq s_comment*
set comment to a sequence. This sequence comment can be extracted with the `Namex(` *seq* `)` command.
Example:

```
 a=Sequence("AFSGDHAGSFDSGAHGSDFASGDA")
 set comment a "a random test sequence"
```

See also: `SEQUENCE.restoreOrigNames`

---

set comp_matrix: redefine residue comparison matrix.

`set comp_matrix` [`add`] *r_increment* [*s_ijPattern*]
change the numbers in the `residue comparison matrix`, called `comp_matrix` by a number typically between 0. and 0.2. This may be very important for generating a reasonable alignment for sequences with low sequence similarity. The result is similar to reducing the `gapOpen` parameter by about 0.1.
Examples:

```
 set comp_matrix add 0.05  # try to Align( ) again
 set comp_matrix 10. "CC"  # make C-C alignment really important
 set comp_matrix add 1. "[KR][KR]"
                           # downweight alignment of Gly against
```

```
                          # all the residues
 set comp_matrix add -.4 "G?"
 set comp_matrix 0. "[AGS][AGSLI]"
```

set directory

`set directory` *s_newDirectory*
change the *current working directory* from inside the icm-shell. We recommend using: `alias cd set directory "$1"` . In this case you can change directory in the Unix/DOS style. Example:

```
 make directory "/usr/tmp" # create a new directory
 set directory "/usr/tmp"
 cd ..    # uses alias from _aliases.
# cd .. is equivalent to set directory ".."
 show Path(directory)
```

See also: `make directory`, `delete directory`, `Path`(directory)

set drestraint

Set a distance restraint between two atoms, or two equal size array of atoms

`set drestraint` *as_atom1 as_atom2 i_DrestraintType* | *R3_low_upper_weight*

Set a distance restraint from interatomic distances

`set drestraint` *distpairs* [*os_ICM*] [*i_cntype*] [`only`] [`find` [`edit`]] [`l_info=no`]

Distances (connections) between two atoms (see `distance`) can be established from the interface or `make distance` command pairs of atoms can be created with a `make distance` command. The convenience of this command is that this object can be easily created interactively and `drestraints` can be directly created based on the atom pairs of this distance-object.

Prerequisites:

- ♦ an ICM object for distance restraints (note that `drestraints` could only be implsed between atoms of the same ICM object)
- ♦ a `distance` object (you can find it in the ICM with the `list parray` command, usually the collection of distances is called `distpairs`)
- ♦ the distances do not need to be between the atoms of the target ICM object. It is sufficient that the atoms mentioned in the *distpairs* object have the same cartesian coordinates as the target atoms (see the `find` option).

Arguments and Options:

Argument	Default	Definition or Comment
distpairs	none	a set of atom pairs, the current distances are not used, just the atoms
os_ICM	`current object` of ICM type	this object must contain
i_cntype	commands finds a type for a close contact between the two atoms	`drestraint type` defining its parameters. Use `show drestraint type` to see the predefined types, set a new type if necessary.
R3_lw_up_wt	sets simple typeless harmonic drestraints	Alternative to the *i_cntype* . Example: 0.//0.//1. or 0.//3.//10.
`only|`	delete all drestraints that previously existed in the object	
`find` [edit]		finds atoms in the specified target object or current object with the same coordinates as the *distpairs* atoms. With the `edit` option ICM requires the source atoms to be between ligand and receptor.
`l_info=no`		

current value in the shell	to suppress the output, you may also use `l_warn=no` to suppress warnings

*Action*Identifies atoms in the *distance* object, finds the same atoms in the *os_ICM* (option `find`) or uses only atom pairs in a_*.LIG molecule and a_REC. object and sets a distance restraint between them. If the type is not specified with the *i_cntype* parameters, the type is found automatically achieve a van der Waals contact between two atoms in question.
Output

- ♦ the `drestraints`
- ♦ `i_out` returns the number of restraints imposed

Option `all` . Set a distance restraint between two groups of atoms (NMR)

`set drestraint all` *as_atomGroup1 as_atomGroup2 i_DrestraintType*
sets distance restraints of specified type between selected sets. Drestraint types (integer numbers) can be either read from a `*.cnt` type file or set directly by the `set drestraint type` command and shown by the `show drestraint type` command.

Setting NMR-style group restraints and with R^{-6} averaging.

Suppose that you have an NMR restraint (with weight 10., and bounds 3. and 4.) between hydrogens belonging to a group, e.g. hb1,hb2 or hb3 of alanine2 and ha1 or ha2 of a glycine10. In this case you can use these commands:

```
read object s_icmhome+"crn.ob"
set drestraint type 1 10. 3. 4.
set drestraint all a_/2/hb* a_/10/ha* 1  # type 1
 # Info> one multicenter (3x2) dist. restraint imposed
show energy "cn"  # gives you the penalty value
set terms "cn"
minimize  # minimizes the multi-center restraint
```

Option `all` allows you to generate a multicenter restraint. Later, the penalty of this restraint will be calculated by finding an averaging the inverse six powers of all possible cross-distances between the two groups.

Two methods for averaging are available, see the `cnMethodAverage` preference.
Important: Drestraints can only be imposed on **real** atoms, the virtual atoms such as `vt1`,--vt2 are ignored in the `cn` calculation, therefore the `set drestraint a_1//vt1 a_2//vt2 5` command is INCORRECT.
Examples:

```
 set drestraint a_/15/ca a_/18/ca 5     # distance restraint of type 5

 set drestraint type 2. 4. 5.; set drestraint i_2out a_/15/ca a_/18/ca
    # define new type (i_2out) and set it
```

set drestraint type

`set drestraint type` [*i_DrestraintTypeNumber*] *r_WeightingFactor r_LowerBound r_UpperBound* [`local` *r_Sharpness*]
creates a `distance restraints type`. Drestraint types (integer numbers) can also be read from a `*.cnt` type file and command and shown by the `show drestraint type` command.

If the type number is not specified, it is set automatically and returned in `i_2out` .
Examples:

```
# type 11, weight 10., bounds [1.,3.]A
 set drestraint type 11 10. 1. 3.

# local type, sharpness 5.
 set drestraint type 12 10. 1. 3. local 5.

# automated type
 set drestraint type 10. 1. 3. local 5.  # returns in i_2out
 set drestraint i_2out a_/2/ca a_/4/ca
```

set group column

`set group column` *tableColumn* [`off`]

this command is applied to a sorted column in a table changes the view of a table. All the rows with identical cell values for this column are merged into families and the **right arrow** click is enabled to rotate over the the family members. Use option `off` to disable this mode.

Example:

```
group table t {1 2 2 3 3 3} {1.1 2.2 3.3 4.4 5.5 6.6}
set group column t.A  # watch the result in GUI, use arrows
```

set hydrogen : re-calculating coordinates of hydrogens from the connected heavy atoms

`set hydrogen` [*as*]

This command does not create hydrogens, it takes the existing hydrogens and re-calculates their cartesian coordinates from the corresponding heavy atoms.

Warnings: the hydrogen placement by this command is not optimized (see `minimize cartesian`). The previously optimized positions of hydrogens may be moved to sub-optimal positions by this command. This command is best used to create reasonable initial positions for hydrogens after the heavy atom coordinates are re-set.

See also: `set atom` , `build hydrogen` .

set site

`set site` [`only`] *seq I_positions s_siteString* [`type="SITE"`]

`set site` [`only`] *seq s_swissprotSiteString*

`set-site` [`only`] {*ms*|*seq*} [*seq_from* [*ali*]]

`set-site` [`only`] *ms* `swiss` # `find a_P uniprot parent sequences and use them`

`set site` [`only`] [`display`] *rs s_siteString* [`label=0-4`] [`type="SITE"`]

`set site distance` *ms* [*r_siteArrowLength* (`0.`)]
set site to with the specified positions and comment. The default action is `append` . Option `only` erases all site information before setting a new one.

If the string is specified, create a new site according to the provided legal site string *s_siteString* (e.g. "FT ACT_SITE 15 15 Catalytic residue"). The format of the site string is the same as in the swissprot sequence entries. The list of legal site types is given `in the Glossary`.
The site residues in objects can be delete with the `delete site` command and selected with the `a_/F` *`SiteCodes`* selection, (e.g. `a_/FAB` selects residues involved in binging and active site).

Option `label=` sets local `SITE.labelStyle` . Value 0 means 'unset'.

The **distance** option allows one to set the length of the site arrow. The default is zero. Caution: the `set site distance` command will re-set all site arrow lengths in a current molecule.

Example:

```
 read sequence s_icmhome+"s.seq"
 set site sss "FT ACT_SITE 15 15 active site residue"
 set site sss {10,15,16,17} "Site1: active site"
 # the residues of this site can be selected as a_/F"Site1*"
#
 read pdb "2abx"
 readUniprot "NXL1A_BUNMU"
 set a_a swiss "NXL1A_BUNMU"
 set site a_P swiss
```

See also: `copy site`, `delete site`, showsite{show site} and `color site`.

set site alignment

`set site` *ali* {*icol*(`1`) [,*jseq*(`1`), [,*ncol*(`1`),[*nseq*]]]} [`column`[=*I_cols*]] [`comment=`*s*] [`type=`'`SHADE`'|'`BOX`'|'`FNT`'|'`FNT_BLD`'|'`REGION`'] [`color=`..]

annotates a region in the alignment.

Example:

```
  set site alig column={4,5,6,7,8} type="REGION" comment="text"  # sets upper region an
  set site alig {10,2,5,5} type="BOX" color=red  # draw the box at row=2, col=10 size=5
```

Example (annotate binding sites)

```
  read binary s_icmhome + "example_alignment.icb"
  set site alig column=Index( alig, Sphere( a_H [1] a_A 4. ) ) type="BOX" color=red
```

See also: `delete site alignment`

copy site

`copy site` [`only`] { *seq_to* | *ms_to* } *seq_from* [*ali*]
transfer (or reassign) `sites` from a sequence or string to a destination sequence *seq_to* or a `selection of molecules` *ms_to* . Sites are listed in feature tables of swissprot entries and are read by the `read sequence swiss` command.
If alignment is not provided, the sequences will be automatically aligned to find residue-residue correspondences and the reliability of the alignment will be reported. If the source of sites is not provided the sites will be transferred from the sequences `linked` to objects. The list of sites and their one-letter codes is given `below`. Normally this command appends to the list of existing sites, unless the `only` option is given in which case the old sites are dismissed.

The effort is made to avoid repetition and retain only the unique set of sites. Identical site will not be added, e.g. simply repeating the same `copy site` command will not duplicate the number of sites.

Example:

```
 readUniprot "PIM1_HUMAN"
 read pdb "1xws"
 make sequence a_1.1
 a=Align(PIM1_HUMAN,1xws_a)
 copy site PIM1_HUMAN 1xws_a
 Info> 8 sites (i_out) appended to 1xws_a
 copy site PIM1_HUMAN 1xws_a  # repeat
 Info> 0 sites (i_out) appended to 1xws_a
```

See also:

- `site`
- `set site`
- `show site`
- `SITE.appendStyle` (`"none"` or `"merge` source")

set site to a residue selection

`set site` [`only`] *rs s_sideString*
assign sites to a molecular 3D object (simpler than the previous Swissprot-like definition).
Example:

```
 read object s_icmhome+ "crn.ob"
 set site a_/10:13 "candidates for mutagenesis"
```

set slide

`set slide` name *slideArray s_oldname s_newname*

Rename object names referenced in a slide array. Useful when an object is renamed after making a slide.

Example

```
nice "1crn"
add slide
rename a_1crn. "crambin"
display slide index=1
set slide name slideshow.slides "1crn" "crambin"
display slide index=1
```

See also: `slide`

set tautomer

`set tautomer` *ms i_tau*

`set tautomer` *rs_his i_tau_1_or_2* | `"hid"` |`"hie"` |`"hip"`

switches between different tautomers of small molecules *ms* or histidine *rs_his* by relative tautomer number or histidine tautomer name. The states and necessary hydrogens are built/set by the `build tautomer` command.

Example:

```
build string "AHW"
build tautomer a_/his # adds a hydrogen and hydrogen masks to allow the switching
set tautomer a_/his 2
```

See also: `build tautomer`

set texture

`set texture` *grob imageArray*

updates textures used in the grob. Textures should be in the order provided by the `Image` command. Common usage would be: get textures, modify them in ICM, and assign them back to grob.

See also: `Image`

Example:

```
read grob "g.obj"
I = Image( g texture )
I = Image( I 256 256 ) # rescale all images
set texture g I # update images used for textures
```

set error

`set error`

sets the icm-shell error flag. The flag is returned (and cleared) with the `Error`() function. Example:

```
if Nof(Getarg(name))==0 set error
a=Getarg("t",2)
if Error() then
  print "Help"
endif
```

set field by number or name

Each object, molecule, residue or atom have a place to store numbers. This place is called a `field` and has a reference number. In addition, atoms have *named* fields that can store numbers or text. Also, user fields can be stored in sequence alignments (see the last section of this page)

Setting a named field in molecular objects

`set field name=` *s_fieldName as|rs|ms|os* { *r|i|s_FieldValue* | *R|I|S_arrayOfValues* }

See the description `below` , as well as the `Field` and `Select` functions.

Setting field in molecular objects by number

`set field` *as|rs|ms|os* { *r_FieldValue* | *R_arrayOfValues* } [`number=` *i_fieldNumber*]
`set field clear` *as|rs|ms|os* [`number=` *i_fieldNumber*]
set user-defined values to atoms, residues, molecules or objects selected. Atoms have one user-field, residues have three, molecules and objects have sixteen. To specify which field you need to set, use the `number=` option.
To extract the property use `Field` (*selection*, *i_fieldNumber*) function.

Level	Max.Nof_fields	example
Atom	1	`set field a_//c* Mass(a_//c*)`
Residue	3	`set field a_/trp 1. number=2`
Molecule	16	`set field a_W Random(1.,10.,Nof(a_W)) number=12`
Object	16	`set field a_*. Rarray(Count(Nof(a_*.)))`

User defined fields can further be 2D or 3D averaged with the `Smooth` function and selected by with the `Select` function.

Setting a field in an alignment

`set field` *ali i_vectorNumber* [*R_aliPosValues*] [*s_name*]

Stores `rarray` of values for each position of the alignment into field *i_vectorNumber* (an integer from 1 to 3) of the alignment. Each alignment has 3 reserved vectors. These values can be used in `set color alignment` command.

The *R_aliPosValues* can be calculated set for each position of the alignment or assigned from sequences via the `Rarray( ali seq R_prop )` projection function.

See example in `set color` *alignment* See also: `set field` *as* `name=` *s*

set atomic field from a map

`set field` *map* [*as*] [`name=`*s_field_name*]

sets the interpolated value from a map to an atom according to the coordinates of its center. Example:

```
loadEDS "3pah" 0. # loads m_3pah crystallographic 2Fo-Fc map for epinephrine
read pdb "3pah"
set field m_3pah
set field a_// name="eds" Field(a_//)
display
set label atom a_// Sarray(Iarray(100.*Field(a_//)))
display ball Select(a_// "eds<0.4" )
center a_aale
```

set named field

`set field name=`*s_name as|rs|ms|os* { *i_value|r_value|s_value|I_values|R_values|S_values* }

Example:

```
set field a_//o* name="Occ" Occupancy( a_//o* )
Field( a_//o* "Occ" )
Name( a_// field ) # returns {"Occ"}
```

This field can be manipulated with the following commands and functions

- ♦ `Field` (*as* | *rs* | *ms* | *os s_tag*) returns the field value
- ♦ `Select(` *as* | *rs* | *ms* | *os s_tag*) returns the atoms for which the field is set
- ♦ `Select(` *as* | *rs* | *ms* | *os s_tag_condition*) returns the atoms/residues/molecules/objects for which the field is set and the condition is met
- ♦ `delete field` [*as* | *rs* ..] *s_tag* deletes the field from some atoms or all the atoms by default
- ♦ `Name(` *as* | *rs* | *ms* | *os* `field`) returns a unique list of assigned tag names at the appropriate selection level

Example 1:

```
 build string "ala"
 set field name="my" a_//c* Count(Nof(a_//c*))  # set values 1,2,3,.. to carbons
 Select( a_// "my" )
 Select( a_// "my==1" )
 delete field name="my"
```

Named fields with text

```
build string "AHW"
set field name="na" a_//n*,c* Name( a_//n*,c* )  # store some atom names in field named
show Field( a_//n*,c* "na")   # returns the value
Select( a_// "na" )  # select atoms with that field set, namely n*,c*
Select( a_// "na==n" )  # select atoms with that field equal to "n"
Select( a_// "na~ca*" ) # fuzzy comparison
Select( a_// "na~c*" )
```

Action upon double clicking an atom .

An action can be assigned to a field with a fixed name `doubleClick` . The atom selection for the action should be coded as dollar-1 ($1). A simple action can be just like that:

```
set field a_// "display cpk $1" name="doubleClick"
```

A toggle can also be easily implemented with a few ICM commands. Example in chick double click toggles the display of bfactors and non-standard occupancies:

```
a1 = "atomLabelStyle=8; if Nof( $1 & a_*.//DA )==0 then; display atom label $1 ; else;
set field a_// a1 name="doubleClick"
```

It may be move convenient to write the toggle expression in a macro, e.g.

```
macro toggleBfactorDisplay as_
  atomLabelStyle=8
  if Nof( as_ & a_*.//DA )==0 then
    display atom label as_
  else
    undisplay label as_
  endif
endmacro
```

and them use the macro in

```
set field a_// "toggleBfactorDisplay $1" name="doubleClick"
```

set font

`set font` [{ `atom` | `residue` | `variable` }] [`auxiliary`] [`bold`] [`italic`] [`underline`] `size=`*i_Size* `font=`*s_FontName*

set current font for atom-, residue-, variable-, or string- labels in the graphics window. Strings can be displayed in either their main font or the auxiliary one (option `auxiliary`). The following fonts: times, helvetica, courier and symbol, should be available. Default fonts are defined in the `icm.clr` file.

Examples:

```
 set font 28 times        # 'Times' font, size 28
#
 build string "se his"
 atomLabelStyle="[C]"
 display wire atom label
 set font atom 14 bold    # for atom labels
#
 set font auxiliary bold italic symbol 28
```

specifying the font in ICM

A few ICM commands use similar parameters to specify the font used in graphics window:

[`bold`] [`italic`] [`underline`] `size=`*i_Size* `font=`*s_FontName*

Font families supported by the `font` option:

"mono", "courier"	a standard monospace font
"serif", "times"	a standard serif font
"sans", "arial", "helvetica"	a standard sans serif font
"symbol"	font with special symbols and Greek letters

See also: `set font`, `set font grob`, `set label 3d label`, `display string`.

set font of a 3D label

`set font` *g_label* [*font_spec*] [*color_spec*]

sets/resets font for a particular 3D label (technically it is a `grob` with a single point and associated text).

See `font specification format` and `color format` for explanation of the *font_spec* and *color_spec* parameters.

Example:

```
#label3d = Grob("label",Mean(Xyz(a_/3,4)),"3D label for res 3,4")
label3d = Grob( "label", {0. 0. 0.}, "HELLO WORLD!" )
set font label3d font="times" size=36 rgb="#00ffdd"
display label3d
select edit label3d # makes it movable, press Esc to get rid of the cursor
```

Make an alignment , an html document, or a table active.

`set foreground` *s_htmlVarName* | *aliName* | *tableName*

Bring the specified GUI panel for the foreground, i.e. make an alignment , an html document, or a table active.

Examples:

```
set foreground s_html s_anchor
set foreground ali
set foreground tab
```

`set foreground center` {`table`|`html`|`alignment`|`graphic`}

Brings the specified class of GUI objects into the central part of the main window

Example:

```
display new  # creates an empty 3D window
add column t {1 2 3 } # creates a table
set foreground center table   # moves table pane to the center
```

See also: `Name(` `foreground` `table|alignment|html`) to get the names of those shell objects

set format for a table column

`set format` *I|R|S|P_column* [*i_width*] [*s_format*|`html`|`web`] [`function=`*s*] [`filter=`*s*] [`name=`*s_cname*] [`color=`*s*] [`show` [`off`]]

`set format` *T_table ..same args and action on ALL COLumns..* [`table`|`view`|`grid`] [`show` [`off`]]

`set format` *I|R|S|P_column k_collectionFormat* # `eg` .. `t.A Collection(t.B format)`

Set various display and auto-calculation properties for table column or all columns of a table. All the set fields can also be extracted into a single collection with named members, modified and reset back to a table of table columns.

- `width` - column width in pixels
- *s_format* - string containing html tags formating for the column cell. Use %1 for reference to a cell value. E.g: "**%1**" display values as bold. For real values the number of digits can be adjusted using "%.n[fge]" format. Where `n` is precision, 'f' - decimal notation, 'g' - exponent notation, 'g' - mixed. E.g: "%.2f" - two digits after dot. The *s_format* string may contain internal ICM html links (see `gui programming`} which allows one to bind any custom action to them.
- `name=s_displayName` use custom name as column name on GUI
- `color=s_colorSpec` a conditional expression which can used for custom cell coloring. (see example below) The expression has the following syntax: `condition` ? `result_if_true` : `result_if_false` `result_if_true` and `result_if_false` themselves can contain conditions (be recursive) The condition may use number and string constants as well table column names. E.g: MW < 100 The returned values should be a string containing either name of the color ('lightred') or html hex notation ('#BAFFBA')
  - ◊ option `rainbow=`*color1*[*/color2***...**][*,from:to*] (e.g. `set format t.A color="rainbow='red/white/blue,100:150,linear/0:0/0.7:0.5/1.`
- `show off` hide column
- `show` show hidden column
- `filter=s_columnFilter` a conditional expression which is used for row filtering. (see example below)

Example:

```
add column t Chemical({ "CCC", "CCO" })
add column t Mass( t.mol ) name="MW"
set format t.MW "<b><p align=right>%.4g</p></b>"
set format t.mol 150
set format t.MW color = " MW>45 ? 'red' : 'green' "
# add an external color column
add column t { "#BAFFBA" "#FFCACA" } name="clr"
set format t.clr off # hide it
set format t.mol color="clr"  # color by clr column
set format t.MW show off  # hide column

set format t.mol filter="_ ~ 'O'"  # show only containing oxygen
```

The following example show how to bind any custom action to table cells.

```
# create a random table
makeTable Name( "t" unique ) 10 2 1 0 no yes yes no
# set action to simply print cell content
set format t.A "<!--icmscript name=\"1\"\n print %@.%^[%#]\n--><a href=#_>%1</a>"
# bind a simple dialog and action.
set format t.B "<!--icmscript name=\"1\"\n#dialog{\"AAA\"}\n# i_n (1|2|3)\n print %@.%^
```

See also:

- `set property column`
- `Collection` (*t*| *col* `format`)
- show-format*t*|~~*col*

set grob coordinates and string label

set *g_grobName M_Xyz*

set *g_grobName* [append] *s_Label*

set *g_grobName* reverse # reverse grob normals so that the light is from inside.

set grob selection reverse

set grob selection [append] *s_Label*
Set new coordinates to the vertexes of the specified graphics object. The matrix dimensions should correspond to the number of vertices. The initial coordinate matrix can be extracted with the Xyz (*grob*) function.

```
read grob s_icmhome+"beethoven" # try stravinsky if you want
display beethoven
display "DESTRUCTION OF CLASSICAL MUSIC"
xyz= Xyz( beethoven )
fuzz = Random(-0.2,0.2,Nof(xyz),Length(xyz))
xyz = xyz + fuzz
set beethoven xyz
color beethoven Random(Nof( beethoven ),3, 0., 1.)
```

Invert grob normals
set grob [selection] reverse
change direction of vertex plane normals in *all* grobs to change direction of lighting and sign of the Volume function. If option selection is specified only the GUI-selected grobs are processed.
set *g_grobName1 g_grobName2* .. reverse # obsolete. Now 'grob select'
change direction of normals in *specified* grobs. In some simple grobs the order of vertices defines the normal implicitly. In this case the order is changed.
An example in which we contour a density map, split the grob into outer shell and cavities and measure their volumes:

```
read pdb "1est.m/"
make map potential 1. Box( a_ )
make grob m_atoms 0.2 exact solid
split g_atoms
set grob reverse   # invert normals of all grobs
Volume(g_atoms1 )  # outer shell is now illuminated from inside
Volume(g_atoms2 )  # cavities have now positive volume.
```

set key

set key *s_keyName s_Command*
binds key to a command. Allowed keys: F1, .. F12, Ctrl-F1, .. Ctrl-F12, Ctrl-A, ... Ctrl-Z, Alt-A, ... Alt-Z. Add "\n" at the end if you want your command to be automatically executed.
Examples:

```
set key "F10" "set plane 1"
set key "Ctrl-B" "l_easyRotate=!l_easyRotate"
set key "F11" "varLabelStyle=\"nextItem\"\n"
```

set label

set label *as_atomForResidueLabels*
assign residue labels to the selected atoms as_atomForResidueLabels . The atoms at which the labels are displayed can be returned with the **L** selection in the atom field, e.g. **a_a.b/10:24/L** .
Examples:

```
build string "se trp ser ala tyr"
set label a_/tyr/cb  # move label from Ca's to Cb's for all tyrosines
```

set label distance

`set` [`residue`] `label` `distance` *rs* [{ *R_3displVector* | *M_displMatrix* }]
reset the relative displacements of the selected residue labels *rs_* to their default of the specified positions. If vector is specified, all the relative displacements are set to this vector, if a relative displacement matrix Nx3 is given, each selected label is moved to the specified relative position. The default position is the relative displacement of {0. 0. 0.} from the residue label carrying atom (usually the Ca atom for peptides, also see the `set label as_` command). See also: `GRAPHICS.resLabelDrag`
Examples:

```
build string "YYEAH"
set label a_/tyr/cb  # move label from Ca's to Cb's for all tyrosines
display a_*  residue label
GRAPHICS.resLabelDrag=yes  # now drag labels with the MiddleMB
set label distance a_/2:4  # reset labels for residues 2:5
set label distance a_/2:4 {1. 0. 3.}
```

---

set labels for table rows

`set label` *T_table i_label* [`index=I_indices`]

Assigns to table rows. Row labels are used to highlight table rows in GUI and for scripting purposes.

Example:

```
group table t {1 2 3} "A"
set label t 1 index={1,3}
Label(t)
```

See also: `Label` `Index` `table label`

set labels for 2D chemicals

`set label` *chemarray* [`S_labels`|`s_label`] [`color=s_color`] [`distance=r_dist`] [`index=I_`]

Assigns annotation (sites) for selected atoms in 2D chemical spreadsheets. Atom selection can be done using `select chemical` command.

Example:

```
add column t Chemical( "COc1cc(C=C2C(N(CC(O)=O)C(=S)S2)=O)ccc1OCc1ccc(cc1[Cl])[Cl]" )
select chemical t.mol filter="AtomNum==1"
set label t.mol "First Atom"
select chemical t.mol filter="AtomNum==2"
set label t.mol "Second Atom"

select chemical t.mol "C(=O)[O;H]"
set label t.mol "Carboxy" color="red" distance=1.5

select chemical t.mol off
```

See also: `select chemical` `delete label chemical`

set label for 3D labels

`set label` *3Dlabel* [`selection`] [*s_text*] [*color_spec*] [*font_spec*]

change the text label properties for a *`3Dlabel`* `object` . Changes will be only applied to the selected labels in *3Dlabel* if the `selection` keyword is used.

This command may change:

- ♦ the label text *s_text* ;
- ♦ label color *`color_spec`* ;

- ♦ label font properties *font_spec* .

set the current map

`set map` m_theMapYouWantToWorkWith
assigns the `current map` status to the specified map.

---

set the current Molcart connection

`set molcart connect=`*s_connectionID*

Sets the `Molcart` connection to be the current.

`set molcart database` *s_dbname*

Sets the current Molcart database to *s_dbname.*

See also: `molcart`, `molcart connection options`, `connect molcart`

setting names to chemical compounds in an array or a table

`set name` *chem_array* { *S_names* | *s_name* } [[`index=`]{*i_index*|*I_index*}]

assigns specified names to each element of the *chem_array* . This names can be extracted with the `Name(` *chem_array*) function. Example:

```
read table mol "drugs.sdf"
set name drugs 2 "aspirin"
set name drugs.mol {2,25} {"aspirin","cocaine"}
#
n=Nof(drugs)
set name drugs.mol Sarray(n,"drug")+Count(n)
set name drugs.mol drugs.synonim
```

In the chemical tables there is a special column 'NAME_' to acceess chemical names. Normally this column is created automatically created upon reading an `.sdf` file. You can sort, search in the column. All modifications made the 'NAME_' column will be automatically synchronized with chemical names (and vice-versa)

However, if the `_NAME` column is created manually, to convert it into a legitimate and synchronizeable name of a chemical one needs to use the `set name` command.

Example:

```
read table mol "t.sdf" name="t" # NAME_ is created automatically. It will be synchroniz
t.NAME_[1] = "aspirin"
print Name( t.mol[1] )
```

setting names to chemical compounds in an array or a table

`set name` *seqarray* { *S_names* | *s_name* }

Assigns names to elements of `sequence parray`. If array of names *S_names* is specified, it should have the same size as the sequence array.

Setting the current object

`set object` [*os_newObj*] [`stack`]
assigns the `current object` status to the specified object. Switches to the next one by default.

Option `stack` means that the in-object-stack will be extracted from the object into the shell. It is equivalent to the `load stack object` command.

Examples:

```
set object a_crn.   # set it to object crn
set object a_1.     # set it to the first object
set object          # switch to the next or alternative
set object a_2. stack # switch and extract its built-in stack
```

See also: `set type` *os_ s_type* .

set occupancy

`set occupancy` *as_select r_NewOccupancy*
sets occupancy of selected atoms to or by a specified real value between 0.0 and 1.0
Examples:

```
set occupancy a_/2:5/!ca,c,n,o 0.5

set occupancy a_/2:18/ca,c,n 1.
```

set plane

`set plane` [move] [*i_plane*] [{ `off` | `on` }] [`name=` *s_planeName*]
toggles the specified graphics plane on and off. Up to seven planes can be set. Optional name is assigned to a plane. It is a convenient way to operate with complex composite images. Every image is assigned to a certain graphical "plane" when displayed. Different parts of the image can be assigned to different planes. For example, **plane 1** may contain wire representation of molecule1, **plane 2** its molecular surface ("surface") and **plane 3** molecule2 in "xstick" representation. It can be achieved by pressing "F2" and "F3" (which are `aliased` to **set plane 2** and **set plane 3**, respectively) before displaying surface and xstick respectively. Now by pressing "F1" , "F2" and "F3" one can toggle these three screens (or planes) to display any combination of them. It is much better than `undisplay`ing and `display`ing them directly, especially for representations requiring serious computations like `surface` and `skin` . The main modes of the set plane command:

- set plane 2 : if plane2 is 'off', make current and switch it 'on'; if it is 'on', switch it off.
- set plane 3 on : switch the plane on, but do not change the current plane
- set plane 4 name="homologue" : just assign name to the plane, no switching

Examples:

```
build string "se ala ala"  # create a peptide
set plane 2  # F2 with the cursor in the graphics window
display surface
set plane 3  # F3 with the cursor in the graphics window
display xstick
set plane 2  # switch off the surface
set plane 2  # switch the surface back on
set plane 3  # switch off the xstick
set plane 3  # switch the xstick back on
```

set pmf

`set pmf` *I_icmTypes* [`energy=`*r_eDepth*(`-10.`)] [`margin=`*r_maxDist*(`8.`)] [`function=`*i_power*(`2`)] [`delete`]

this command sets the specified potential with the *r_eDepth* value at distance = 0. and 0. at distances beyond ~r_maxDist for the *iIcmType* : *iIcmType* interactions for the types specified in the *I_icmTypes* array. The functional dependence is defined by the `function` argument (the default is a quadratic function). The function is:

```
E =  r_eDepth *( 1. - x/r_maxDist ) ^ i_power
```

For example if you want atoms of type 8 to attract each with a constant force (and linear dependence of the energy as a function of distance) use this:

```
set pmf {8} function=1 delete
```

Arguments and options:

- *I_icmTypes* the `pmf` force field will be assigned between pairs of atoms of the same type from the specified list. We usually prefer to use unused hydrogen types such as 7,8,9,28,29,32:40,44:48 . This will still make the artificial atoms visible (in contrast to the *virtual* atoms) and will not affect any of the "real" atoms. Use `set type` *as iType* command to set the artificial types Check the `icm.cod` file for the available types.
- `delete` : makes sure that the specified types do not interact as van der Waals spheres, and incapacitates those atom types. See the suggested types above.
- `energy` = *r_eDepth* . see formulat above. The value of energy when two atoms of specified type are at the zero distance
- `function` = *i_power* . The exponent of the functional dependence above.
- `margin` = *r_maxDist* . The interatomic distance at which the "mf" term becomes zero.

Example:

```
build smiles "C1=CC=CC=C1.C1CCCC1"
set type a_//h?1 8
set type a_//h?2 9
set pmf {8,9} margin=6. energy=-5. function=3 delete
display
color a_//C8 green
color a_//C9 magenta
show energy "vw,mf"
```

See also:

- pmf
- `show pmf`
- read pmf *file* # e.g. ident.pnf in s_icmhome

set property

`set property` [`only`] [`on` | `off`] *icmShellObject1 .. prop1 prop2 ..*
ICM shell objects of the following types: integers, reals, strings, sequences, alignments, profiles, maps, matrices, tables, grobs, iarrays, rarrays, sarrays have an array of property elements. This elements can be set to `on` and `off` from the shell the they influence visibility, edit-property and some other properties of a variable in the GUI environment.
Allowed property elements:

bit_name	description
`command` *s*	indicates that the string contains ICM commands and is a script
`delete`	protects from the delete command
`display` *T_*	activates `table actions` such as double click, cursor and lock
`field` T_	makes the content of individual cells of a table un-editable
`factor` T_	indicates that the table is a table of structure factors
`html` s_	indicates that the string is an `HTML document`. It may contain internal links to scripts, images and slides
`show`	makes object name invisible in the Workspace, is `off` for system variables
`write`	indicated that object will be written in `write binary all` command. This option is 'on' by default.
`smiles`	indicated that elements of an `sarray` will be treated as smiles string and depicted on-the-fly in the table.

Option `only` resets the property mask to 0 before setting the specified bits. Example:

```
ii = {1 2 2 3}
group table t {1 2 3} "a" {3.3 3.3 4.4} "b"
set property t only  # clean up
set property t ii write delete field off # protect the content
```

More examples:

```
s2 = "read pdb \"1crn\" delete\ndisplay ribbon yellow\n"
set property command s2  # s2 will appear in Workspace
```

set table column options

`set property` *T_column* {`fix`|`field`|`new`|`plot`|`show`} [`off`] [`only`]

- `field` allows one to edit cells
- `fix` freezes a column to always keep in sight during horizontal scrolling through a large number of columns
- `new` marks a column as having a new content (a flag to update a view)
- `plot` : converts cell-vectors into in-cell plots (e.g. `add column t Matrix(3); set property plot t.A`)
- `show` off : hides a column

See also: `set property chemical view`, `set format`

set chemical view options

`set property chemical view` *chemicalColumn s_chemicalProperies* [`only`] [`off`]

sets various chemical view options for the molecular column of the ICM table.

Each character in *s_chemicalProperies* codes single chemical view option.

- "H" : Hetero-atom hydrogens
- "T" : Terminal hydrogens
- "S" : Atom stereo labels
- "X" : Do not show explicit hydrogens
- "A" : Aromatic rings"
- "C" : Show 'chiral/racemic' flag
- "3" : Do not show 3D as 2D
- "U" : Unique atom classes
- "N" : Atom numbers
- "M" : Monochrome atom labels
- "W" : Don't show atom text labels. Colors half of the atom's adjustment bond with the element color (Like wire in 3D)
- "R" : Don't show atom text labels. Draw color square instead.

Example

```
add column t Chemical("CC(=O)OC(C=CC=C1)=C1C(O)=O")
set property chemical view t.mol "HM"      # monochrome labels + hetero atom hydrogens
set property chemical view t.mol "M" off   # turn off monochrome
set property chemical view t.mol "A"       # turn on aromatic ring view
```

set alignment view options

`set property` *alig i_mask* [`only`] [`off`]

sets various view properties for the alignment:

- 512 : do not show consensus line
- 1024 : display tree
- 2048 : show alignment profile
- 8192 : do not show sequence offset
- 65536 : do not show alignment body. Useful if you want to export profile only.
- 524288 : show ruler

Multiple values can be combined used + operator.

Example:

```
set property myAlig 2048+65536   # show profile only
```

set randomize : reset the randomSeed

`set randomize` *i_NewRandomSeed*
resets the random seed to the new value. If you run any procedure or function for the first time, it will show you the value of `randomSeed` . This value can be reset at any time later with the

above command.
Example:

```
Random(1,10)
 Info> randomSeed = 1055822291
 4

set randomize 1055822291
Random(1,10)
 4
```

set resolution

`set resolution` *os* { *r* | *R_NewResolutions* }

set resolution of selected objects to a specified real value or individual values from the *R_NewResolution* array. To assign individual resolution, provide a real array with resolutions for each object.

Example:

```
read pdb "1crn"
print Resolution( a_ )[1]
set resolution a_ 9.9
print Resolution( a_ )[1]
```

See also: `Resolution`

set stereo

`set stereo` [*i_plane*] [{ `off` | `on` }] [`name=` *s_planeName*]
this command allows one to reset the stereo mode from a command line or scripts. See also: `GRAPHICS.stereoMode`

set sstructure backbone

`set` *rs s_SecStructPattern*
assign the specified local secondary structure to the selected residues of an ICM-type object. Note that this command **changes the conformation** of the selected residues, in contrast to the command `assign sstructure.`

The *s_SecStructPattern* string (e.g. "HHH___EEE") can be shorter than the number of selected residues. In this case the pattern will be applied multiple times. For example:

```
set a_/A "E" # will set all residues to an extended conformation
```

The phi,psi angle values are changed according to the following code:

ss_code	phi,psi angles	description
_	-179.9,179.9	extended conformation
E	-139.0,135.0	antiparallel beta strand
e	-119.0,113.0	parallel beta strand
H	- 62.0,-41.0	alpha-helix
G	- 49.0,-26.0	G-helix (3/10)
I	- 57.0,-70.0	I-helix
P	- 78.0,149.0	poly-proline 2 helix
L	+ 57.0,+47.0	Left-Alpha

Examples:

```
 build string "LLELGQAPGALHRVPLSRRESLRKKLRAQGQLTELWKSQNL"
 display ribbon residue labels
 set a_/2:8 "H"   # all 6 residues will be assigned to a helix
 center
 set a_/1:12 "HHHHHH__EEEE"
 center
 set a_/A String("H", Nof(a_/A) )
 center
```

```
set a_/A String("_", Nof(a_/A) )
center  # ONLY UNFIXED PHI,PSI VARIABLES ARE SET, SO pro IS BENT!
set a_/A String("G", Nof(a_/A) )
center
set a_/A String("E", Nof(a_/A) )
center
```

set sstructure to sequence

`set sstructure` *seq s_SSstring* `i_from i_to`
set secondary structure *s_SSstring* to the specified sequence. If *s_SSstring* is an empty string, the secondary structure definition is removed.
Examples:

```
a=Sequence("LLELGQAPGALHRVPLSRRESLRKKLRAQGQLTELWKSQNL")  # 1st seq.
b=Sequence("PLLEATQIKVPLKKIKSIREVLREKGLLGDFLKNHKPQ")     # homologue
set sstructure a "HHHHHHHHHHH______EEEEEEEE_____HHHHHHHHH__"
l_showSstructure = yes
show Align(a b)
```

`set sstructure` *seqarray S_sstructures*

set secondary structure strings *S_sstructures* to elements of `sequence parray`. Array sizes should match.

set stack properties

`set stack` [*os*] `loop`|`fast` [`off`]

`set stack` [*os*] `energy` [*from to*] *R_NewEnergies*

`set stack` [*os*] `number` [*from to*] *I_nVisits*

`set stack` [*os*] `all` [*from to*] *I_nTotalVisits*

resets stack display parameters, energy values, or number of visits, or total number of visits, for conformations stored in the `stack` . If the object is specified, the internal object stack is modified. New energy values may be useful for the subsequent `sort stack` command.

`set stack align` [*from to*]

will set the total visits to 1 and will set the visits to {1,2,3,..}. This setting is convenient since now the visits can be used as and an ID of a conformation while the total visits at 1 is helpful for future compression (the `compress stack` will add up those 'ones' into the total number of conformations compressed into one bin.

If *from* and *to* are not specified, they are assumed to be `1` and `Nof(stack)` .

The stack display parameters.

- `loop` equivalent to the `loop` option in the `display stack` command, it replays the stack until interrupted with the ICM interrupt.
- `fast` option prevents interpolation between stack conformations (the default is 20 interpolated frames)

See also:

- `store conf` command
- `Nof( stack )` function
- `sort stack`
- `compress stack`
- `compare` command
- `display stack`

set swiss

`set swiss` *ms_proteinChains* { *S_swissprotCodes* | *s_swissProtCode* }
set swissprot name (like `IL2_HUMAN`) to one or several chains selected by *ms_proteinChains* . To clear it just set it to an empty string.

E.g.

```
build string "AAAAAAA"
set swiss a_ "SILLY_HUMAN"
Name(a_A swiss)[1]
 SILLY_HUMAN

set swiss a_P ""  # clear all previously set swiss IDs
```

Warning: Uniprot/swissprot may change uniprot ids and they become obsolete. Swissprot IDs are at any given time unique but perishable, while the accession numbers AC are not unique (many different ACs for the same entry) but permanent.

See also: `Name(` *ms_* `swiss`) function.

set crystallographic symmetry group

`set symmetry` *os_object R_6cell s_symgroup* | *i_symgroup* [*i_NofChains*]

`set symmetry` *os s_crysym_card* # `contains "group N Z a b c alpha beta gamma"`
assigns symmetry and cell parameters to selected object(s). The combined `crysym` record is often available in exports.

The set of parameters is be compatible with that provided in **CRYST1** PDB card:

- *R_cell* should be a 6-component real array, containing values of *A, B, C, alpha, beta* and *gamma.*
- *s_symgroup* is a string description of the space group. To check validity of the *s_symgroup*, use the `Symgroup(` *s_symgroup*)} function, which will return a number from 1 to 230 for a valid space group name. Fast Fourier transformations are currently supported for *s_symgroups* "P 1" and "P 21 21 21", but all the other commands (`make map cell transform` etc.) will work on any space group defined in the `International Tables for Crystallography.`
- Z-value, the number of polymer chains in a unit cell, is extracted from the last integer parameter or assigned automatically according to the number of transformations of the symmetry group.

Examples:

```
 build string "se ala ala ala" name="z"
      # suppose this is my modified crambin
 set symmetry a_z. { 40.96 18.65 22.52  90.0 90.77 90.0 } "P 21"
```

---

set biological symmetry to an object

`set symmetry` [append] *ms* `R_12N_transformations`

sets biological symmetry to selected chains of the object. The biological symmetry is applied to **all** the molecules belonging to a certain chain. For that reason it is recommended to use the `molecular selection by chain` (e.g. `a_Cabc` for chains a,b,c) and use the `set chain` command if required to assign one chain character to a group of molecules.

By default, the previous biological symmetry will be overwritten. The `append` option tells the program to add a new biomolecule record.

Example:

```
  read pdb "2ins"
  set chain a_a,b,zn "A"
  set symmetry a_CA Transform( a_ )[13:24]
```

See also:

- ♦ `makeBioMT` macro in the _macro file
- ♦ `Nof( os_1 "bio")` # number of biomolecules
- ♦ `Select( os_1 "bio" i_Biomol )` # molecules of i-th biomol.
- ♦ `Transform( os_1 "bio" i_Biomol )` # transformations

set symmetry to a torsion

`set symmetry` { `1` | `2` | `3` | `6` | `exact` | `heavy` | `pseudo` } *vs*
assigns rotational symmetry to selected variables. This symmetry will be used to automatically transform the value of a torsion angle into [-180.0/symmetry , 180.0/symmetry] range.
Options are the following:

- ♦ exact - impose exact symmetry (methyl groups=3, xi2_phe=2)
- ♦ heavy - impose exact symmetry as if there are no hydrogens
- ♦ pseudo- impose pseudo symmetry (no_hydrogens + xi2(his,asn,gln))

---

set table

`set table t_theTableYouWantToWorkWith`
assigns the `current table` status to the specified table (similar to `set object` *os_* to set the current molecular object).

---

set energy or penalty terms

`set terms` [`only`] [*s_termsString*]
set `energy and/or penalty terms` for further energy calculations. Each term has a two-character abbreviation. The terms are appended to the string unless option `only` is specified. The final energy-term string is returned in the `s_out` string
Examples:

```
                  # vacuum terms, solvation and entropy
 set terms only "vw,14,hb,to,el,sf,en"
 set terms "tz"  # add tethers to the list
```

---

set selftether

`set selftether` [*as* [`only`]] [`tether`|*R_xyz*|*M_xyz*] # `copy x,y,z to selftethers`

`set selftether delete` [*as*]

sets target coordinates for the specified atoms. These positions then can be used as `selftethers`.

Example:

```
build string "AHW"
set    selftether a_//c*
set    selftether a_//n* only # clears the previous ones and sets  nitrogen selftethers
delete selftether        # delete all selftethers in the current object
```

See also:

- ♦ `selftether`
- ♦ `delete selftether`
- ♦ `"ts"` term
- ♦ `TOOLS.tsToleranceRadius` and `TOOLS.tsWeight` parameters.

set tether

`set tether` [`align` | *ali*] [`exact`] [`only`] *as_atomsToBePulled* [*as_atomTargets*]

`set tether residue` *rs_toBePulled rs_targets* # no residue alignment is forced, residues are equivalenced sequentially

set tether *P_atompairs* [*os_ObjToBePulled* }
this command sets tethers restraining atoms of ICM-object (selection *as_atomsToBePulled*) to corresponding atoms of another object (*as_atomTargets*). The *as_atomTargets* selection may also contain only **one** atom, in which case all *as_atomsToBePulled* will be tethered to a single atom. If the second argument is not specified, all the *as_atomsToBePulled* atoms are tethered to the origin (the {0. 0. 0.} point). Option only signals that all previously imposed tethers must be deleted.

The **residue alignment** is controlled by the alignment options .

If option residue is specified, it just takes the selected residue pairs in sequential order.

If parray of atom pairs is specified (it can be created with the make distance command or with the GUI distance tool) the tethers are picked from suitable atom pairs of the specified *P_atompairs* object. If the explicit tethered object is not specified, it is assumed to be the current object .

In a residue pair the only the backbone atoms such as ca,c,n,o,ha,hn are tethered with the exception of

- ♦ identical residues: all atoms are tethered
- ♦ F with Y (all but the hydroxyl)
- ♦ D with N
- ♦ E with Q

The number of imposed tethers is saved in i_out .
See also: superimpose, alignment options, minimize tether.
Example (try this series of commands in one continuous session):

```
build string "se glu ala"  # a simple object
set tether a_/2            # tether to the origin
display tether wire virtual
minimize v_//?vt* "tz"

delete tether
build string "se gln val" name="gv"  # another object
set tether a_2.//ca,c,n a_1.//ca,c,n exact # tether set to set
display tether wire a_*. only
minimize v_//?vt* "tz"

delete tether
set tether a_2.//ca,c,n a_1./1/ca  # tether to a single atom
display tether wire
minimize v_//?vt* "tz"
```

set tether append: Extending the identified substructure with neighboring atoms

set tether append [all]

if maximal common chemical substructure was identified using the find molecule command and tethers were imposed between the matching atoms, the initial set of tethered atoms can be further propagated into the neighboring atoms. Without option all only suitable hydrogens are added to the initial match. With the all keyword heavy atoms will also be added. Note, that any two heavy atoms next to a tethered pair are considered a match and will be paired.

Example:

```
build string "H"
rename a_ "his"
build string "W"
find molecule sstructure tether all a_his.//!h* a_//!h*
set tether append a_  # add single hydrogens
set tether append all a_  # add heavy neighbors
```

set atom type

set type [mmff] [*as* { *i_type* | *I_type* }]

assigns the specified atom type (see icm.cod or show atom type [mmff]) to the selected atoms. Both the ICM- and the mmff- atom types may be manually adjusted to correct the automated set type mmff command.

---

set type property : contributions of atoms types to the property grids.

set type "apolar"|"atomic"|"membrane" *R_sf_density_values_in_kcal_A2*

reset the "atomic solvation" or "apolar" surface based implicit solvation energy densities.

See also: surfaceMethod preference, icm.hdt file containing the default icm values. Example:

```
surfaceMethod = "atomic solvation"
x = { 0.0080,0.0220,-0.0900,-0.2240,-0.1760,-0.0630,-0.0350,-0.2240,-0.0960,-0.1160,\
     -0.0120,-0.0510,0.0080,0.0080,-0.0630,-0.0900,-0.0900,-0.1760,-0.0900,\
      0.0,0.0100,0.0100,0.0100,0.0100,0.0100}
set type "atomic" x
```

set type property : contributions of atoms types to the property grids.

set type property *R_upToSevenWeights* [only] [*I_listOfAtomTypes*]

This command defines the contribution of the listed atom types to each of up to seven grid maps named g1 g2 g3 .. . This grid maps will be used by the "gp" energy/penalty term for local or grobal energy optimization (see show energy , minimize and montecarlo).

Arguments and options

- ♦ *R_upToSevenWeights* provides weights of contributions for this atom type to the grids for the make map potential "gp" command, as well as the maximal contribution that atoms with those atom types will get in g1, g2, .. etc. The number of elements in this array determines the number of grids.
- ♦ *I_listOfAtomTypes* is an iarray of types, e.g. {100,111,112}, or Count (100,199) for a range of types.
- ♦ option only means that for these atom types the weights not covered by the *R_upToSevenWeights* array are set to zero.

The types are listed in the icm.cod file. If the *I_listOfAtomTypes* is not provided, all heavy atoms will be set to contribute to grids.

Example to set different fields for oxygens (types 50 to 99) and all other atoms:

```
set type property {1., 0.} Count(50,99) only
set type property {0., 1.} Count(1,49)//Count(100,390)
```

A better way to set the default types would be to use the setApfTypes macro, e.g.

```
build smiles "C1NCCCC1"
setApfTypes
make map potential "gp"
```

To set the weights of energy contributions from the individual g1, g2, .. modify the gpWeights array of parameters, e.g.

```
gpWeights = {2., 1. , 0., 3. , 2., 1., 1.}
gpWeight = 3.
```

The overall contribution ofthe weighted sum can further be weighted with the gpWeightparameter.

See also:

- ♦ `make map potential` "gp" .. # generating up to seven grid maps.
- ♦ `term "gp"`

see script _chemSuper and _chemAlign .

set object type

`set type` *os s_type*

change the type of one of several non-ICM objects. The following types are allowed (two dots denote the minimal necessary string):

- ♦ `"pharma.."` or `"ph4"` - pharmacophore
- ♦ `"ca"` - C-alpha models only
- ♦ `"xray"` or `"x-ray"`
- ♦ `"nmr"` - solved by NMR
- ♦ `"model"` - general, or generated by modeling
- ♦ `"electron.."` - solved by electron diffraction
- ♦ `"fiber.."` - solved by fiber diffraction
- ♦ `"neutron.."` - solved by neutron diffraction
- ♦ `"simple"` - specialized simple models

Example:

```
build smiles "C1CCCCC1"
strip a_  # can not redefine the ICM type
Type(a_ 2) # check it before
set type a_ "pharmacophore"
Type(a_ 2) # check it after
```

set molecule type

`set type` *ms s_type*

change the type of the selected molecules. The following types are allowed:

- ♦ `"A"` - amino (proteins and peptides)
- ♦ `"N"` - nucleic acids (RNA and DNA)
- ♦ `"H"` - heteroatoms (most of the chemical compounds)
- ♦ `"M"` - metals
- ♦ `"W"` - water
- ♦ `"S"` - sugars
- ♦ `"L"` - lipids
- ♦ `"R"` - radical
- ♦ `"U"` - unknown
- ♦ `"0"` - switch to automated type definition from residue types (returned by the `Type` function)

These types are frequently used in scripts and macros. The types can be selected, e.g. `a_M`,W (metals and waters). Note that function `Type`(ms_1 2) returns the auto type only.

Example:

```
read pdb "2ins"
show a_zn1
   5  zn1           1 zn1   2ins  H _  #   zinc ion on 3-fold crystal axis
set type a_zn1 "U"   # here we reset the type to 'unknown'
show a_zn1
   5  zn1           1 zn1   2ins  U _  #   zinc ion on 3-fold crystal axis
```

set type sequence

`set type` [*seq* | `sequence` | *ali*] { `protein` | `nucleotide` }
assigns the specified type to the sequence (*seq_*), all sequences (`sequence`) or sequences from the specified alignment or sequence group (*ali_*). The type can be returned by the `Type`(seq_) function.
Example:

```
  aaa = Sequence("AAAAATAAAA")
  set type protein aaa

  read sequence "f.seq" group="tmp"
  set type tmp nucleotide
```

set type mmff

`set type` [`charge`] `mmff` [*os*]
automatic assignment of the MMFF atom types for the selected or the current object of any type. This object can be both ICM-object or a non-ICM object, provided three conditions are satisfied:

1. the bond types are set correctly
2. the formal charges are set correctly
3. the object is complete and has hydrogens (see the `build hydrogen` command)

This command is a prerequisite for the `set charge mmff` command (it can also be achieved with the `charge` option).

set van der Waals radii

`set type` "`vw radii`" *I_vwTypes R_vwRadii*
reset radii defined in the `icm.vwt` for *I_vwTypes* to the *R_vwRadii* values. The van der Waals radii are used for the surface calculation in the `show surface area`
command

set electrostatic radii

`set type` "`vwel radii`" *I_vwTypes R_vwRadii*
reset electrostatic radii marked as vwel defined in the `icm.vwt`. The electrostatic radii are used in the `boundary element electrostatic` calculation.

set 3D view rotation, translation and size

`set view` *R_37ViewVector*

`set view` *R_37FinalViewVector nMilliSeconds*

`set view` *R_37InitViewVector R_37FinalViewVector nMilliSeconds*
sets all the parameters of the graphics window (position, size, zoom, rotation, etc.) according to a `rarray` of 37 numbers. If the *nMilliSeconds*parameter is specified this command makes a smooth transition between two views. The first view is either the current view or the *R_37InitViewVector* view. The final view needs to be specified explicitly.

This array is returned by the `View` () function and can be created, read and written as an ordinary real array. Aren't you disappointed that you still do not know the meaning of these parameters? It is dull, believe me, use the command and take it easy. See also: `View`, `rotate view`.
Example:

```
 read pdb "1crn"
 display a_1crn. ribbon  # now move the molecule, resize window ..
 write View( ) "a.view"  # write 37 numbers in a file
   # again: rotate, move/resize the window etc., or quit the session
 read rarray "a.view"    # read 37 parameters
 set view a              # restore the view
```

set vrestraint

`set vrestraint` [`energy`] *rs* [*s_rsTypeName1 s_rsTypeName2* ...]
sets variable restraints of specified types to the selected residues `rs_` . Variable restraint type names (strings) can be read from a `*.rst` type file and shown by the `show vrestraint type` command. Option `energy` enforces the "energy" type of vrestraint.
Number of imposed variable restraints is saved in `i_out` .
Examples:

```
 set vrestraint a_/*            # assign all zones to relevant residues
```

```
set vrestraint a_/ala "aa" "ab" # assign alpha and beta zones to all Ala residues
```

set vrestraint variable

`set vrestraint` [`only`] [{ `energy` | `fix` }] *vs r_1 r_2* [*r_3*] [*R_values*] [`name=` *s_rsName*]
impose a set of `vrestraints` to the specified variables *vs_*. The zone will be a multidimensional elliptical well *around current values* (default), or the specified *R_values* values, of the selected variables. The shape of the well in each dimension is a *soft square well* . Three types of vrestraints can be imposed, depending on the option:

- **probability** vrestraints (the default). They are marked as "rs" in the `icm.rst` file. Probability vrestraints are used in the BPMC procedure to define the distribution of random steps. The well parameters are as follows:
  - ◊ *r_1* : *r_relProbability* , the relative probability of this vrestraint
  - ◊ *r_2* : *r_wellRadius*, the well radius

  The relative probability is in arbitrary units, it is only important as a relative number in a *group* of the vrestraints.
- **energy:** "Energy" vrestraints (marked as "rse" in the `icm.rst` file). These allow the formation of the multidimensional wells around groups of variables and are used to softly restrict the variables to certain zones (see the `"rs"` energy term). The well parameters are as follows:
  - ◊ *r_1* : *r_energyDepth* (it must be negative for attractive wells)
  - ◊ *r_2* : *r_fractionFlat*
  - ◊ *r_3* : *r_wellRadius*

  Parameter *r_fractionFlat* (between 0. and 1., default 0.) defines flat fraction of the energy well for the energy vrestraints. **Note:** one can create both **wells** and **bumps** using negative and positive values of *r_energyDepth*, respectively Example:

  ```
  build string "se nter ala ala cooh"
  set vrestraint energy v_/3/psi -20., 0.2, 200., # WELL OF DEPTH 20.
  set vrestraint energy v_/3/psi  20., 0.2, 200., # BUMP OF HEIGHT 20.
  ```

  An example from the `_dock2mol.icm` script: imposing an individual restraint for the `virtual` bond:

  ```
                  # no penalty for deviations up to 15A
  set vrestraint energy v_2//bvt1 only  -50.0 0.5,  30.0
  ```
- *R_values* contains target values for each angle in the selection *vs_* , e.g. {-120.,60.}, By default the target values are taken from the current values of the selected variables.
- **fix:** Vrestraints on "fixed" variables (marked as "rsr" in the `icm.rst` file). These are used to define switches between different fixed conformations, e.g. alternative conformations of sugar rings, proline rings, switches between L and D amino-acids etc. These switches will be tried in the `montecarlo` procedure if these variables are included in the set of vs_MC variables but not included in the set of the minimization vs_min variables. The parameters are defined as follows:
  - ◊ *r_1 r_relEnergy*, relative energy of a conformer
  - ◊ *r_2 r_relProbability* .

  The *r_relProbability* is in arbitrary units as for the probability vrestraints. Example with L-D transition, through changing the sign of the two phase angles:

  ```
  build string "se ala his trp"
  set vrestraint fix V_/3/fha,fcb  Value( V_/3/fha,fcb ) 0. 1. name="l"
  set vrestraint fix V_/3/fha,fcb -Value( V_/3/fha,fcb ) 0. 1. name="d"
  montecarlo V_/3 v_//*
  ```

The radius of the vrestraint well (in degrees for angles) is given by the *r_wellRadius.* Option `only` deletes all the previous vrestraints. The name is optional. The names of the "probability" and "fix" vrestraints are be shown in the output of the `montecarlo` procedure. The names need not be unique.
Example: creating a file with equal probability vrestraints around stack conformation angles with 30 deg. radius:

```
read stack "f1"      # read conformational stack
for i=1,Nof(conf)    # go through all the conformations
  load conf i        # load them one by one
  set vrestraint v_/2:5/phi,PSI,xi1  1. 30.
endfor

build string "se ala his trp"
```

```
set vrestraint v_/2/phi,xi1,xi2 ,{-60.,-60.,120.} 0.5, 45. name="bb"
set vrestraint v_/2/phi,xi1,xi2 ,{ 60.,-60.,120.} 0.5, 45. name="cc"
montecarlo v_/2/phi,xi1,xi2
```

Note that in the command a special `PSI` torsion specification is used for traditional residue attribution.

---

set values of internal coordinates

`set` *vs* [`add`] { *r_value* | *R_arrayOfValues* }
sets specified variables to a given value(s) (for angles the value must be in degrees). If `rarray` *R_arrayOfValues* is specified, its values are assigned sequentially to the variables. It the array is shorter than the selection, the values are applied periodically. Option `add` means increment by the specified value rather than set to this value.
Examples:

```
read object s_icmhome+"crn.ob"
set v_//phi -60.                 # all phi to -60 degrees
set v_//phi,PSI { -60., -40. }   # make sure that the first
                                 # variable in selection is phi

set v_/1:8/phi Random(-180.,180.,8) # all different random phis
set v_/1:8/phi add 2.0              # increase 8 phi angles by 2 degrees
```

Note that in the second command a special `PSI` torsion specification is used for traditional residue attribution.

---

set positional variables to place a molecule to polyhedral vertices

`set` *vs* `grid` *i_vertex i_NofVertices*
(order of arguments is important!) sets specified 2 variables (normally a `virtual` planar angle and torsion angle) to the values such as to put a molecule in the vertices of tetrahedron (i_NofVertices=4), octahedron (6), cube (8), icosahedron (12) or dodecahedron (20). Used to sample uniformly the surface of globular molecules. Values of i_NofVertices other than above are not allowed. The polyhedron is built around the origin. The size of the polyhedron is determined by `v_//bvt1` variable which is a `virtual` bond length from the origin to the first virtual atom (vt1) of the two attached to each molecule. To check how polyhedrons are generated look at this example:

```
read object "complex"
display virtual a_//ca,c,n | a_//vt* only
color molecule
set a_1//vt1       # set vt1 of a_1 to its center of mass
set a_2//vt1       # set vt1 of a_2 to its center of mass
set v_1//bvt1 0.1  # move a_1 to the origin (0.1 to avoid a singularity)
set v_2//bvt1 30.  # offset a_2
                   # this is for a_2 to hop around a_1
for i=1,20
  set v_2//avt1,fvt1 grid i 20
endfor
                   # this is for a_2 to rotate need the same location on a_1
for i=1,12
  for j=1,3
    set v_//avt2,tvt3 grid i 12
    set v_//tvt2 j*120.
  endfor
endfor
```

---

set size and position of ICM graphics window

`set window` [*i_xLeft i_yDown*] *i_xSize i_ySize* [`margin=` *r* ...] # without GUI

`set window full` [`on` | `off`]

`set window fix` { *i_xSize i_ySize* | `off` } # `with GUI`
sets the position and/or size (only size if 2 arguments are given) of the graphics window without Graphics User Interface (use option `fix` otherwise). Four arguments are in pixels. If you need to display in a fixed size window from a script we recommend to use the set window command first and then the `display` command.

The `full` option will switch into the fullscreen mode (also Ctrl-F and Esc to switch off) This option does now work with GUI.
In the off-screen mode (see the `display off` command) `set window` is accompanied by re-`centering` of the molecular image with `margin=` *r_* ... and other `center` options.
The `fix` option will change window size for ICM in the GUI mode. In this case the window may become smaller than the actual area in the master GUI window. Option `fix` is used to make video clips with ICM using fixed size frames.
Example:

```
            # square 700x700 window in the upper left corner
 set window 570 30 700 700
 display window
 set window 300 300
 write image window=3*View(window)  # hi-res. image
```

set xstick radii

`set xstick` *as_select r_NewRadius* | *R_matchingArrayOfRadii*
sets occupancy of selected atoms to or by a specified real value between 0.0 and 2.5A . See also:

- `GRAPHICS.stickRadius`

show

`show` *args* [`output=`*s_outputStringName*]
show information about specified ICM-shell objects in your shell-window. Show is similar to the `list` command, but it gives you more information, covers a broader range of subjects and allows the user to show constants, subsets and expressions. However, in contrast to the `list` command, `show` does not understand **wildcards.**

Option `full` will show arrays and shell variables which are grouped into tables (the components of tables are hidden by default). The same option `full` temporarily sets `l_showSpecialChar` to `yes` when sarrays are shown.

Option `output` allows one to dump the result into an ICM string variable with the specified name for further analysis.

show selftether

`show selftether` *as*

shows atoms with `selftether` restraints imposed (require the "ts" energy terms to be activated in `minimize` or `montecarlo`) The show command also returns the number of selftethered atoms (`i_out`), the number of deviating atoms (`i_2out`) and the maximal deviation in `r_out`

See also: `selftether`

show site

`show site` [*ms*] [*seq_1 seq_2..*]
show `sites assigned` to the selected molecules *ms_* or sequences. By default all the sites of the `current object` are shown. See also: `set site`, `color site` .

show shell variable

`show` *arg1 arg2* ... [`output=`*s_stringVarName*]
show `ICM-shell variable`, `constant`, `subsets`, or `expressions`. One needs to separate arguments by comma only if two consecutive arguments are numbers, and the second on is a negative number constant. Option `output` allows one to dump the result into an ICM string variable with the specified name for further analysis.
Examples:

```
 read alignment msf s_icmhome + "azurins"
 show azurins[3:20]   # show a fragment of the alignment
 show a b a*b         # two arrays and their product
 show Sin({1. 3. 5.}) # another array
 show 2., -3.         # without the comma, it will show -1.
 show m_crn           # map (m_crn) header information and
                      # the map sections
```

show key

show key
show commands bound to key-strokes. Allowed keys: `F1, .. F12, Ctrl-F1, .. Ctrl-F12, Ctrl-A, ... Ctrl-Z, Alt-A, ... Alt-Z`. See also the `set key` command.

show map

show { map | mapName }
show the current or the specified map in text format. Example:

```
 build string "AKSD"
 make map potential Box(a_) "ge"
 display m_ge {1 2 3 0 4 5 6}
 show m_ge
 m_ge> written in ZYX mode (z-sections). Symmetry group #0
    Box  {sect0,row0,col0, sect,row,col} = {-30,-8,-21, 32,16,28}
    Cell {A,B,C, angles(deg)} = {14.000,8.000,16.000, 90.00,90.00,90.00}
    Nof intervals (at x,y,z)  = {28,16,32}
    Min/max/mean/rms density  = -20.000000, 20.000000, -0.182712, 12.082560
 ...
 ::::::::::::::::**##########
 ::::::::::::::::**##########
 ::::::::::..:::**##########
 ::::*****::..::::**#########
 :::***###*:..::::***########
 ::***######**...:::****#######
 :***######**...::::****######
 :***######*:...::::******###*
 :***##**::........:::::********
 :*****:::.......:::::********
 :****:::.........::::*******
 ::***:::.........::::*******
 ::***.............::::******
 ::***:.............::::******
---{13 / 32}-    # shows pages
```

show objects, molecules, residues, atoms and variables

show { *os* | *ms* | *rs* | *as* | *vs* }
show selected atom(s) `as_`, residue(s) `rs_`, molecule(s) `ms_`, object(s) `os_`, or variable(s) `vs_`, respectively.
Examples:

```
 show a_*.      # all objects
 show a_*.*     # all molecules of all objects
 show a_2.*     # all molecules of the second object
 show a_*       # all molecules of the current object
 show a_/ala    # all alanines of the current object
 show a_1//c*   # carbons of the 1st molecule of the current object
 show v_2.a//phi,psi
```

Data fields for **objects** :

```
show object
 # a_objectName.  type    n_Mol  n_Res  n_waters resolution  object_name
  1  a_def.  Type: ICM    Mol: 1 Res: 4    def
  2  a_1dna. Type: X-Ray Mol: 3 Res: 532 Wat: 216 Resol: 2.20 thymidylate synt..
```

These fields can be accessed with the following functions:

- object name: `Name` (*os_*)
- object type: `Type` (*os_* , `2`) # returns "X-Ray","NMR","ICM",etc.
- number of molecules: `Nof` (*ms_*), e.g. `Nof( a_2.* )`
- number of residues: `Nof` (*rs_*), e.g. `Nof( a_2.*/* )`
- resolution: `Resolution` (*os_*), e.g. `Resolution( a_2. )`
- number of waters: `Nof` (*water_selection*), e.g. `Nof( a_2.w* )`
- full name: `Namex` (*os_*), e.g. `Namex( a_2. )`

Data fields for **molecules** :

```
read pdb "1a36"
show a_*
    Name     n_residues first_res_name  object_name
--{i Molecule}- N_Res                Object ---
   1  a            544 ile               1a36
   2  b             22 dpa               1a36
   3  c             22 dpa               1a36
   4  w1             1 hoh               1a36
   5  w2             1 hoh               1a36
...
```

These and other molecule attributes can be accessed with the following functions:

- mol. name: `Name` (ms_)
- mol. type: `Type` (ms_ , `2`) # field not shown Returns. "Nucl","Amino","Hetatm" etc.
- number of residues: `Nof` (*rs_*), e.g. `Nof( a_2.*/* )`

---

show alias

`show aliases`
show all currently defined aliases. To show a specific alias, use the
`alias` *aliasName*
command (e.g. `alias cd`).

---

show alignment

`show alignments` [`color`]
show currently loaded alignments. Option `color` colors residues in the alignment by type.

---

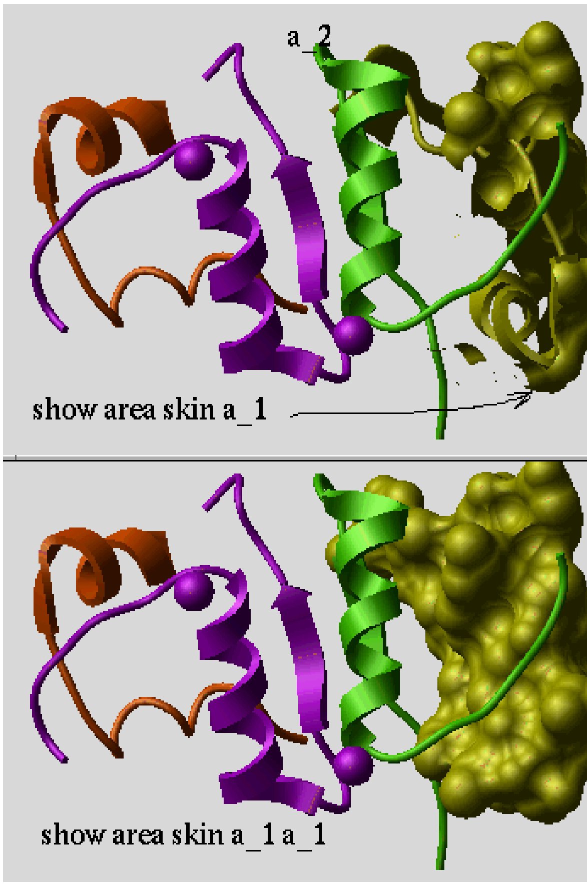
show area

`show area` { `surface` | `skin` } [`mute`] [*as_1* [*as_12*]] [`surfaceAccuracy=` *i_level*] [`waterRadius=` *r_newRadius*]
Calculates the area of the solvent-accessible `surface` or molecular surface (so called `skin`), respectively. The probe radius is defined by the `waterRadius` parameters (1.4 by default). You can specify for which atoms you want to calculate the surface (selection *as_1*). The `surfaceAccuracy` level defines the 'resolution' of the surface calculation. The default level is 3 but the level of 5 is recommended for if the surfaces are used to make a decision about the atom burial. You can also additionally specify the environment for these selected atoms, i.e. the neighbors which you want to take into account in the surface calculation.
The two most popular modes are the following:

- measuring the surface area of some atoms being a part of the whole system (e,g, `a_1 a_*` or just `a_1` , the top picture)
- measuring the surface area of a group of atoms as if they are the only atoms that exist in space (e.g. `a_1 a_1` the bottom picture).

In essence, two optional selections [*as_1* [*as_12*]] impose a mask on atom pairs, so that only pairs in two selections are considered. If only the first selection is specified, the second one is assumed to be *all atoms* . The two *reasonable* choices for the second selection are *all atoms* (the default), and the *repetition* of the first selection (acts as if not other atoms are present in the system). In all cases, the second selection must include the atoms of the first one, e.g.

```
 show area skin a_1 a_1,2 waterRadius=1.2
```



The total area will be stored in `r_out` and the number of triangles used in the "skin" construction in `i_out` .

The individual areas are stored with atoms and can be returned with the `Area(` *as_* `)` function. Warning. This command only fills out the values for the selected atoms. If you want to set the values of other atoms to zero, use the -{set area a_//* 0. } command. Example:

```
read object s_icmhome+"crn.ob"
set area a_1//* 0.  # make sure that the initial area is zero
show surface area a_1//!h* a_1//!h* # only the first molecule
show Area(a_//*)   # individual areas, hydrogens have 0.
show Sum(Area(a_//!h*))  # the total
```

show atoms

`show` *as*
shows properties of the selected atoms. Example:

```
build string "se ala"
show surface area
show a_//c*
 Atom Res    Mol Obj   X       Y       Z     Occ    B  MMFF Code   Xi Chrg formal Grad Area
   ca   1   ala  a1 def -2.748  0.000 -2.245 1.00   20.0   1 113 C  1   0.06  0     0.0  0.5
   cb   1   ala  a1 def -2.329 -1.202 -3.093 1.00   20.0   1 113 C  0 -0.09  0     0.0  7.3
   c    1   ala  a1 def -4.247 -0.000 -1.935 1.00   20.0   3 121 C  0   0.45  0     0.0 34.2
```

The fields:

Field	Description
`Atom`	atom name
`Res`	residue number+[symbol] and name
`Mol`	molecule name
`Obj`	object name

X,Y,Z	coordinates
Occ	occupancy (from 0. to 1.)
B	B-factor (positive)
MMFF	MMFF atom code
Code	ICM atom code
Xi	chirality number (0,1,2,3)
Chrg	partial charge
formal	formal charge
Area	solvent accessible surface area
Grp	electrostatic group (atoms can not be separated)
as_	selection expression

show atom type

`show atom type` `show atom type mmff` [{ *s_pattern* | *i_type* }]
shows atom types stored in the `icm.cod` file. The `mmff` option allows one to check the Merck Force Field atom type.
Examples:

```
 show atom type
      # show all ICM types
-------------{atom codes}-----------
#
#     icd   vw   hb   hd      wt      sf na
#
atcd    0    0    0    0   0.000    0.00 ?
atcd    1    1    1    0   1.008    0.00 h
atcd    2    3    1    0   1.008    0.00 h
...
 show atom type mmff "*cation*"
      # cations
 show atom type mmff "*iron*ion*"
      # do we have iron ions?
 show atom type mmff "?C=*"
      # what types are connected to doubly-bonded carbon ?
 show atom type mmff "[!C]*ring*"
      # non-carbon types in rings
 show atom type mmff 32
      # some oxygens
-----------{MMFF atom codes}--------
 Symb.Typ.[V] Description  {formal charge}

  O2CM 32 [1] oxygen in carboxylate anion
  OXN  32 [1] N-oxide oxygen
  O2N  32 [1] nitro oxygen
  O2NO 32 [1] nitro-group oxygen in nitrate
...
```

show bond : detecting problematic covalent geometry

`show bond` *as* [`mute`|`error`|]

goes through all bonds of the selected atoms (returned in `i_out`) and does the following:

- checks the number of bonds per atom, counts atoms with more than four bonds
- finds bonds shorter than 0.6A and longer than the sum of two van der Waals radii multipled by 0.7. Counts bonds that are two short or too long
- reports the number of problematic bonds or bond numbers in `i_2out`

show clash

`show clash` [`mute`] [*as_1* [*as_2*]] [*-r_vwDistanceFraction*] [*r_distance*]
shows all the interatomic distances between two atom selections which are shorter than the sum of two van der Waals radii multiplied by the *r_vwDistanceFraction* parameter (0.8 by default). This command can be shown to show the *short* contacts only if the limit is about 0.8, or show show all pairs of atoms with significant van der Waals contribution (the limit of about `1.2`)

IMPORTANT: this will work only for the ICM-objects.

Use the `show energy "vw"` command (and pay attention to the current fixation) to pre-calculate interaction lists. The output will show the actual distance and the ratio of this distance and the sum of radii. Mark the two atoms of interest, separated by a logical OR, and paste it into another command if necessary.

The number of van der Waals contacts satisfying the *r_vwDistanceFraction* criterion is returned in the `i_out` shell variable.

The **mute** option suppresses the screen output (`i_out` is still calculated).
See also: `display clash`, `undisplay clash`. Visualize the strained atoms with `show a_//G` or `display a_//G` .
Example:

```
build string "se ala his trp glu"
randomize v_//*
display
show clash a_//c* a_//c*       # clashes between carbons
show clash a_//c* a_//c* -0.7  # more tolerant test
display clash
```

show color list

`show color` [`mute`]
shows list of colors defined in the file `icm.clr` and stores the output list in the `S_out string array`. Option `mute` suppresses output to the screen but still saves to the `S_out` array (useful for scripts)
See also: `color` command.
An example:

```
show color
-------------{colors}-----------
  1 black              #000000
  2 white              #ffffff
  3 grey               #878787
  4 blue               #0065ff
  5 red                #ff0000
...
```

Example of `show color mute` use in a script:

```
if (Exist(view)) then  # check if graphics is active
  show color mute      # saves a list of colors in S_out
  for i = 1, Nof(S_out)
    color background $S_out[i]
    pause
  endfor
endif
```

show arrays as parallel vertical columns

`show column array1 array2` .... [*s_fileName*] [`separator=` *s_Separators*] [`comment=` *s_Comment*]
shows several arrays in a multi- `column` format. If you want to shorten the significant digits in real arrays, use this trick:

```
a = {1.333333 2.44444}  # creating some dumb arrays
b = a
show column Rarray(a,2), Rarray(b,1)
```

See also: `write column`, `show database`, `write database`.
Example:

```
resnam = {"ala" "glu" "arg"}
reschg = { 0., -1., 1.}
show column resnam reschg
show column separator=":" comment="Example table"  resnam reschg
```

show comp_matrix

`show comp_matrix`
shows `residue comparison matrix` used by the `alignment` algorithms.
See also: `set comp_matrix`, `read comp_matrix`.

show table in database format

`show database` { `table` | `array1 array2` .... }
shows several arrays or a table in a database format.
See also: `read database` `show column`, `write database`.
Example:

```
resnam = {"ala" "glu" "arg"}
reschg = { 0., -1., 1.}
show database resnam reschg
```

show drestraint

`show drestraint` [*as_select* [*as_select*]] [`center`] [`mute`] [*r_violation*]
shows distance restraints. Arguments:

- optional *`as_select`* atom selection arguments specify atom pairs to be considered. **Attention**, the `as_out` selection can not be used as an argument since it is redefined by the command.
- *r_violation* : if the *r_violation* distance is specified, only the restraints deviating from the upper or lower bounds by *r_violation* are shown.
- `center` : If `center` option is specified the violation is measured with respect to the target value of the distance restraint and optionally only the distances greater than *r_violation* are reported.
- `mute` option: allows one to fill out the `as_out` selection and calculate the number of selected drestraints (`i_out`) without actually reporting them. It is useful for scripts.

Output:

- `as_out` atomic selection of all atoms for which the specified criteria have been satisfied
- `i_out` reports the **number** of selected drestraints

See also: `drestraint` and `drestraint type`.

show drestraint type

`show drestraint types`
shows available `drestraint types` as defined in the `icm.rst` file. The numbered global or local types can be used to impose `distance restraints`. The other types are fixed and are used to impose `disulfide bonds` or `peptide bonds`.

show energy

`show energy` [`mute`] [*s_termString*] [*vs*] [*as_select1* [*as_select2*]]

`show energy atom` [`mute`] [*s_gridTermString*] [*as_select1*]
calculates and shows values of currently set or explicitly defined in *s_termString* `energy terms` (e.g. "`vw`,el")

If the `show energy atom` option (described below) is used the result is stores it in the bfactorfields with the offset of +20. If *vs_* selection is specified, only the selected variables will be unfixed. The initial fixation will be restored after completion. Two additional atom selections may specify a subset of atom pairs that should be considered by the minimization procedure. Note that the contribution from the `"14"` energy term is not displayed separately. It is included in the `"vw"` contribution. If you want to display it separately, use the more straightforward `Energy("14")` function.
Important: the boundary element electrostatics is the most computationally heavy term. It is activated if electrostatic term `el` is switched on and preference `electroMethod` is set to `"boundary` element" . The most demanding part is the calculation of the boundary and its

characteristics. Therefore, for multiple calculations with the same boundary we recommend to use `make boundary` and `delete boundary` commands.

show energy atom, crystallographic electron density energies

`show energy atom` *os_icm*

calculates individual atomic **grid** energies for the some grid terms. (Note: A more direct way of computing the projected map values on atom centers is given by the `set field map` command.)

Maps used by the the `show energy atom` command:

- `"gc"` (needs `m_gc`) vw heavy atoms
- `"gh"` (needs `m_gh`) vw hydrogens
- `"ge"` (needs `m_ge`) electrostatic
- `"gs"` (needs `m_gs`) hydrophobic
- `"gp"` (needs `m_g1`, ...) properties

the result is added the value of 20. and is set to the atomic bfactorfield (see `Bfactor(` *as*) and set-factor.

Example with the "gp" property field:

```
build string "ASD"
make map potential "gp"
show energy atom "gp"
gp_e = Bfactor(a_// ) - 20. # atomic energy contributions, -20 to eleminate shift
add column t Group( gp_e , a_// "sum" ) Name( a_/ ) full)  # Group aggreates into resid
show t
```

Example with a crystallographic electron density map.

An electron density map needs to be transformed into an evenly spaced orthogonal map with the `make map potential` *m_xray R_box* | *as*command. Example showing how somebody messed up epinephrine's chirality:

loadEDS "3pah" 0. # loads m_3pah crystallographic 2Fo-Fc map for epinephrine read pdb "3pah" # unconverted pdb bx = Box(a_aale 5.) # R_6box around epinephrine convert Res(a_//* & bx) # carve out region of interest and convert to ICM make map potential m_3pah bx # box around epinephrine, makes m_xr m_g1 = Trim(m_xr, -1., 1.) set type property {1.} Count(50,300)//Count(330,404) only # without H set bfactor a_//* 0. show energy atom "gp" set bfactor a_//* & bx 20.-Bfactor(a_//* & bx) Select(a_// "b

See also: `set field map`

show energy gradient

`show gradient`
show gradient calculated by the `minimize` or `show energy` commands.

show hbond

`show hbond` [`mute`] [*as_1* [*as_2*]][*r_maxHbondDistance*]
calculates and outputs the list of `hydrogen bonds` between two atom selections. By default calculation is done between all the atoms of the current ICM object. The real argument *r_maxHbondDistance* defines the upper bound of the distance between a hydrogen and a potential hydrogen acceptor to place the pair to the hydrogen bond list. Default value of *r_maxHbondDistance* parameter is 2.5 A. Number of identified hydrogen bonds is saved in `i_out` . To display/undisplay hydrogen bonds, use `display hbond` and `undisplay hbond` commands. Hydrogen bonds can also be calculated by the `minimize` and `show energy` commands provided that the `hydrogen bond term` is switched on.)

The number of hydrogen bonds satisfying the *r_maxHbondDistance* criterion is returned in the `i_out` shell variable.

The **mute** option suppresses the screen output (`i_out` is still calculated).

show hbond exact : accurate bonding energy calculation

`show hbond exact`
calculate the hydrogen bonding energy according to the distributed electron density geometry. Used in virtual screening to evaluate a score.

---

show table in html format

`show html` *T* [`link` *T.S_1 s_linktype1 T.S_2 s_linktype2 ...*]
show the *T_* table with HTML tags. Interpret web links according to the web link types described in the `WEBLINK.DB` array.
See also:
  ◊ `write html` *s_file T_* [`link` ...] - write the html document to a file
  ◊ `web` *T_* [`link` ...] - directly show the table in the web browser.
Option `none` suppresses the table title and the copyright notice.
Example:

```
show html SR link SR.NA2 "PDB"
```

---

show iarray

`show iarrays`
show integer arrays defined in the shell. It shows names, dimensions and the first elements of arrays. The `I_out` array contains the output of some functions and commands and is always in the shell.

```
ii={1 2  3 4 5 6 76}
iii=Count(10)
show iarray
---------------{iarrays}-------------
   [1:1]    { 0, ... }
   ii[1:7]       { 1, ... }
   iii[1:7]      { 1, ... }
```

---

show integers

`show integers`
show all integer shell variables. Example:

```
show integer
---------------{integers}------------
   a                111
   autoSavePeriod   10
   defSymGroup      1
               0
   minTetherWindow  20
   mnRemarks        3
   mnSolutions      50
  ...
```

---

show label

`show labels`
show graphics string labels to find out their number. Then the labels can be addressed as `label 1, label 2` etc.
See also: `display` *string_label*

---

show library

`show libraries`
show loaded `ICM-libraries`. It's a lot of stuff, enter 'q' to exit.

---

show link

`show link` [*ms*]
show links between molecules of 3D molecules and corresponding sequences and alignments.

---

show logical

`show logicals`
shows all logical shell variables in ICM-shell. Example:

```
aa=yes
show logical
---------------{logicals}------------
  aa               yes
  l_alignProfiles  yes
  l_antiAlias      yes
  l_antiAliasGLfix no
  l_autoLink       yes
  l_bpmc           yes
...
```

---

show mol

`show mol` *as_select*
shows selected atoms in the `mol` file format. See also: `read mol` and `write mol`.

---

show mol2

`show mol2` *as_select*
shows selected atoms in the `mol2` -file format (file extension .ml2). See also: `read mol2` *"file"* and `write mol2` *"file"* .

---

show molecule

`show molecules`
shows all molecules of all objects currently in icm-shell. This command is identical to `show a_*.*`

---

show object

`show objects`
shows all molecular objects currently in icm-shell. This command is identical to `show a_*.`
The same result is achieved with the `list a_*.` command.

---

show pdb

`show pdb` *as_select*
show selected atoms in the PDB file format.
See also: `read pdb` *"file"*, and `write pdb` *"file"*.

---

show pmf

`show pmf`

shows currently set distance functions between pmf types. See also: `set pmf` and pmf

show preferences

show preference
shows all icm preference variables in icm-shell (e.g.

```
 show preferences
 ..
  atomSingleStyle  = "tetrahedron"
        1 = "tetrahedron" # current choice
        2 = "cross"
        3 = "dot"
 ..
```

show profile,rarray,real,sarray,string

show profile | rarray | real | sarray | string
shows all objects of specified type(s) in icm-shell. E.g. E.g.

```
 show sarray rarray
```

show residue

show residues
shows all residues in all molecules of all molecular objects. This command is equivalent to

```
 show a_*.*/*
```

show residue type

show residue types
show names and characteristics of compounds described in the `icm.res` and user ICM residue libraries.

show segment

show segment [*ms*]
show `segment` representation of 3D structure of a protein for the selected molecules *ms_* (all molecules of the current object by default).
See also `assign sstructure segment`, `ribbonStyle`, `display ribbon`.

show sequence

show sequences [selection] [number] [{ fasta | swiss | pir | gcg | msf }]
show all sequences or the specified sequence *seq_* in one of specified formats. The default format is the `fasta` format. Option `number` defines if the residue numbers are added. Option `selection` only shows sequences selected graphically or with the `select sequence` .. command
Three logicals: `l_showSstructure`, `l_showSites`, and `l_showAccessibility` control the display of a corresponding additional information aligned with the sequence.
Example:

```
 readUniprot "RXRA_HUMAN"
 show sequence swiss RXRA_HUMAN

 read pdb "1lbd"
 show surface area
 make sequence
  Info> sequence  1lbd_a  extracted
 show 1lbd_a  # you see relative accessibilities in 0-9 scale
 l_showAccessibility = no
 show 1lbd_a
```

show stack

`show stack` [[*i_FromConf*] *i_ToConf*]
show the following parameters of the conformations currently residing in the `conformational stack`.

- ◊ iconf - a slot number
- ◊ ener - total energy as calculated before the conformation was stored
- ◊ rmsd - the distance (either Cartesian or angular RMSD) between the current conformation of the object and the stack conformation calculated according to the `compare` command.
- ◊ naft - the number of visits AFTER the last improvement of energy
- ◊ nvis - the total number of visits to this slot; since new conformation are only compared with the last stack conformation the conformations may drift and cover a large area than described by the `vicinity` parameter

show table

`show` [`table header`] *T_table* [`database`]
shows the specified `table` in the ICM `table` format (one line per table row) or ICM `database` format (a list of column-name column values pairs for each entry). The `header` option suppresses the column subtitles.

If you want to shorten the significant digits in real columns, use this trick:

```
add column t {1.333333 2.44444}  # creating some dumb table with one column
t.A = Rarray(t.A 2)  # will trim to 2 sign digits
show t
```

See also: `show html` *T_* . Database index tables are exceptions, show T_index will show all the entries of the related database. To see members of an index table type the index table name and press TAB.

show terms

`show terms` [`all`]
shows the active energy/penalty terms. With option `all` it shows all the terms available. The result is saved in the `s_out` string. You can also use the `Info` (term) function to return the term string. See also: `set terms`, `Info` (term), `delete terms`.

show tethers

`show tethers` [`mute`] [*as_select*] [*r_minDeviation*]
Shows tethered atoms with deviation larger than *r_minDeviation* (0. by default) and returns these atoms in `as_out` . Option `mute` is used when you just want to get a selection (as_out) of strongly deviated atoms.
See also: `display tethers`.

show version

`show version`
show characteristics of the current ICM executable. Part of this string containing the version number is returned by the `Version( )` function.

show vrestraints

`show vrestraint` [*vs*]
shows `vrestraints` imposed on the internal variables of ICM molecular object.

show vrestraint type

`show vrestraint types`
shows types of `vrestraints`. These types are loaded from the `icm.rst` file.

---

show volume

`show volume skin` [`mute`] [*as*]

`show volume surface` [`mute`] [*as*]
Calculates the volume confined by the solvent-accessible surface or molecular surface (so called "skin"), respectively . One optional selection *as_1* defines atoms for which the volume is calculated. If the selection is not specified, the atoms are assumed to belong to the current object. The volume will be stored in `r_out` and the number of triangles used in the `skin` construction in `i_out` .
Examples:

```
read obj s_icmhome+"crn.ob"
show volume surface             # inside accessible surface
print "volume inside accessible surface = ", r_out
show volume skin                # inside molecular surface
print "volume inside molecular surface = ", r_out
```

---

calculate volume of blobs of map density.

`show volume` [`map`] [*I_indexBox*`[1:6]`] [*r_Threshold*]
Contour electron density map at a given *r_Threshold* and calculate the volume of the high-density blobs. Defaults:

◊ take the `current map`;
◊ contour the whole map;
◊ use threshold value from the ICM-shell real variable `mapSigmaLevel` .

Threshold is expressed in the units of standard deviations from the mean map value, i.e. 1. stands one sigma over the mean. The volume will be stored in `r_out` . See also: `make grob` *m_* .
Examples:

```
read map s_icmhome+"crn.map"     # load m_crn map
show volume m_crn 3.            # calculate volume inside the
```

---

show supported pharmacophore types

`show pharmacophore type`

lists types of pharmacophoric centers and corresponding SMARTS expressions.

See also: `find pharmacophore`

---

sort

a family of `sort` commands (sort objects, molecules in object, array/arrays or sort tables by their columns).
sort array(s)

`sort` [`reverse`] [`number`] [`history`] `sort_key_array` [`array2 array3` ...]
sort one or several integer, real or string arrays. The first array is the sort key. By default ordering is lexicographic for string arrays and by increasing arithmetic value for integer and real arrays.
Options:

◊ reverse: reverse the sense of comparisons.

◊ number: enforce sorting according to arithmetic value for string arrays.
◊ history: save the old order in **I_out** (new[i]==old[I_out[i]])

See also: `Sort` . Examples:

```
 a={3 2 1 5 7 4 6}
 b=Sin(a*50.)
 c={"three" "two" "one" "Five" "Seven" "four" "Six" }
 show column a b c
 sort a b c
 show column a b c
 sort reverse b a c
 show column a b c
 sort c b a
 show column a b c
```

sort table

`sort` [`reverse`] [`number`] [`history`] `table.keyArray1` [`reverse`] `table.keyarray1` [`reverse`] ...
this command sorts all the arrays of the `table` so that all the listed *table.keyArrays* are applied sequentially with descending priority. Each array can be followed by the `reverse` option to change the sorting order.
Examples:

```
 read table s_icmhome+"res.tab"  # residue properties
 RES = $s_out  # create an ICM table RES
 sort RES.aa             # resort entries by residue name
 show RES
 sort reverse RES.flexInd RES.aa
 show RES
 sort RES.hPhobInd  RES.flexInd
 show RES
```

sort table column

`sort column` *tab* [`function` = *s_expr*] [`reverse`] [`name` = *S_cols*] [`selection`]
this command sorts table columns by name or by custom function/expression

Options:

◊ `reverse` : option to change the sorting order
◊ `selection` : option sort only selected columns
◊ `name` : specify `sarray` of column names to sort
◊ `function` : specify the function or expression to calculate sorting key. (See `add column function` for detailed description of available functions)

Examples:

```
makeTable "t" 10 0 0 3 no no no yes
sort column t function="Icm::Min(COL)"   # Sorts by minimum value ('COL' refers
sort column t name={"B","C"} function="Icm::Corr(COL,A)" reverse  # Sorts column
```

sort and reorder molecular objects

`sort object` *os_ i_pos* # move selected objects to a give position

`sort object R_key|I_key [reverse]` # reorder objects by an array, e.g. `sort object Mass(a_*.)`

`sort object S_key [reverse] [number]` # option number interprets the string array as numbers

`sort object [field` = *i_Field*`] [reverse]`
resorts **all** molecular objects by the specified user field (see the `set field` command,

and the `Field` function). If the `field` is not specified, the objects are sorted by their mass.

sort molecules in an object by mass or a user field

`sort` *os_ObjectSelection* [`field` = *i_Field*]
resorts the molecules in each of the selected non-ICM objects by the specified user field (see the `set field` command, and the `Field` function). If the `field` is not specified, the molecules are sorted by molecular mass. An ICM object can be stripped, resorted and then `converted` again.

Sorting a stack of conformations

`sort stack`
sort `conformations` in a `stack` according to their energies. New energies can be assigned to the same conformations with the `set stack energy` command.

---

split

can split `grobs`, tables into individual components, hierarchical data tree into clusters and DNA/RNA sequences (or protein) by multiple-N stretches.

split grob

`split` *g_complexGrob* [*s_rootGrobsName*] [*i_maxNofGrobs*] [*r_minNofPointsInGrob*]
divide disconnected parts of a `graphics object` into a bunch of separate graphics object sorted according to their size (measured as the number of vertices). The maximal number of new grobs is defined either by *i_maxNofGrobs* explicitly or by the MnGrobs parameter. The latter can be redefined in the `icm.cfg` configuration file. The *i_maxNofGrobs* option allows one to retain only larger pieces. Grobs will be sorted according to their number of points and named by adding their sequential number to the input grob name or *s_rootGrobsName*, if specified.
The split command is used in protein `cavity analysis` and other applications where one needs to treat, display, and measure disconnected parts separately. You can also limit the **number of points** of the grobs generated by the command by providing the real argument with the minimal number of vertices you want in a grob.
See also: `Volume(` *g_)*, `Area(` *g_)*, `Xyz(` *g_)*.
Examples:

```
 read object s_icmhome + "crn"
 make grob skin a_//cb a_//cb name="g_crn"
 split g_crn
 display grob smooth # display as one smooth surface
 undisplay g_crn
 color grob unique
 show Volume(g_crn3) Area(g_crn3)

 read map s_icmhome + "crn"
 make grob
 split g_crn "blob" 30    # create up to 30 largest grobs and
                          # call them "blob1" "blob2"...
# a variant: split g_crn "blob" 40 100.0  # discard grobs smaller than 100. vert
 delete g_crn
 display grob
 color grob unique
```

---

split group : derive replacement group arrays from a combinatorial library and a scaffold.

`split group` *scaffold.mol combilib* [`auto`]

an operation inverse to the `enumerate library` command. In this case we take the library with a common scaffold, specify the scaffold and output an array of replacement groups `R1` , `R2` ...

With `auto` option no explicit R-group specification is needed. The command will automatically find attachment positions and create appropriate columns. Columns which are invariant (no changes of substituents) will be exclcuded.

Example:

```
smi = {"C1CCC2C(C1)CCCN2", "CCC1CCCNC1C1CCCCC1", "CC1CCCNC1C1CCCCC1", "C1CCC(CC1
add column t Chemical( smi )
split group t.mol Chemical( "C1CC(C(NC1)[R2,H])[R1,H]" ) name="tt"
```

See also: `enumerate library`,`make reaction`,`Replace chemical`, `Find chemical`, `SAR analysis`

Splitting a table to arrays

`split` [`t_tableName`]
split `table` into individual arrays.
Example:

```
 group table t {1 2 3} "a" {2 3 4} "b"  # t.a t.b arrays
 split t                                # a and b arrays
```

Splitting a sequence to domains between NNN. runs

`split` *sequence_with_NNruns* [*i_minlen_of_Nrun*]

the sequence will be divided into smaller sequences between NNN.. runs. By default even a single N is a separator. Nowever one can specify the minimal length of the N-run as the second argument. Example:

```
a=Sequence("AAANNAAAAAAAAAAAAAAANNNNNNAAAAAAAAAAANANA" nucleotide )
split a 3
show sequence
```

Splitting multiple values in each cell of a column into single-value cells by multiplying rows.

`split` [*tableColumn*] [`separator=` *character*]
takes each string of the specified column and splits it by the separator (comma is the default separator, e.g. `separator=","`) The rows are multiplied accordingly. Example:

```
group table t {1,2} {"a,b,c","d,e"}
t
 #>-A-----------B----------
    1           a,b,c
    2           d,e

split t.B separator=","
t
 #>-A-----------B----------
   1           a
   2           d
   1           b
   1           c
   2           e
```

Note that extra columns are appended to the original table (that explains somewhat strange order).

Splitting an object into separate molecules

`split` `object`
There is no such command, but if you want to split a molecular object into separate molecules, you can simply copy the object and delete unwanted molecules in each copy. Example:

```
 copy a_ "b"
 delete a_b.!1  # delete all but the first molecule
 write a_b. "b" # contains only the first molecule
#
 copy a_ "c"
 delete a_c.!2  # delete all but the second molecule
 write a_c. "c" # contains only the second molecule
#etc..
```

Changing the position of tree cursor (separator) and calculating new cluster numbers

Rows of a data table or a chemical table can be organized into a hierarchical tree which is stored in the *table*`.cluster` array of the table header. This can be done with the `make tree` command which also creates a column with cluster group indices. The name of that column can be obtained with the `Name(` *table*`.cluster` *i_cluster* `split )` function. The tree can be used to determine clusters at different distance levels.

The threshold distance at which the clustering is made can be reset with the

`split` *table*`.cluster` *i_cluster r_newSplitDistance*

command. This command also recalculates the cluster numbers.

E.g.

```
 split T.cluster 1 0.14 # take the 1st tree and set distance threshold to 0.14
```

See also `Split function`

sprintf

`sprintf` [`append`] *s_formatString arg1 arg1 arg2 arg3* ... [`name=` *s_outputStringName*]
Print to the `s_out` string, or the *s_outputStringName* specified after the `name=` option. The same syntax as `printf` command, but the result is not displayed.
Example in which string `outStr` is the destination:

```
 sprintf "mncalls = %d\n",mncalls name="outStr"
```

store

store things to internal memory structures.
store conf

`store conf` [*i_slotNumber*] [*os_obj*] [*s_comment*]

`store conf` *i_slotNumber* { *r_energy* | `number=` *i_nOfVisits* } [*os_obj*] [*s_comment*]
store current conformation into specified slot of the `conformational stack`. By default it puts the conformation into the first free slot, or appends it to the end. The energy, by default, is automatically extracted from the previous energy evaluation, or taken from *r_energy* if explicitly provided. The total number of visits (`nvi`) is set to 1 by default.

if the *os_obj* argument is provided the conformation will be added to the local stack in the object.

Example:

```
 build string "WSD"
 montecarlo           # generates a stack
 show stack
 set v_//omg 180.     # change a conformation
 store conf -9. "mycomment"      # add conformation with energy -9. and comment
 store conf 3, -9.    # override slot 3 with energy -99.
 store conf number=33 # set conf with number of visits=33
```

See also `set stack` *property array_of_values* command , e.g.

```
set stack energy Random(0., 10., Nof(stack))
```

for multiple assignments of energy values, number of visits or total number of visits.

If *os_sel* argument is provided the conformation will be stored into a object's stack (see also `store stack` *os_* to move the whole stack to the object).

See also: `store stack` *os* to copy the global stack to an object

store conformational stack inside an object

`store stack` *os*

takes the current stack and stores it in a compressed form inside the specified object. The compressed stack can then be extracted with the `load stack object` command. Option `stack` of the montecarlo command stores the generated stack inside the current object automatically.

See also:

- ◊ `delete stack` *os*
- ◊ `copy` *os* `stack`
- ◊ `load stack object`
- ◊ `load conf`
- ◊ `montecarlo` .. `store`
- ◊ `set object` .. `stack`
- ◊ `Exist` (*os1* `stack`)
- ◊ `Nof` (*os1* `stack`) # returs the number of conformations in a stored stack

store frame

`store frame` [`write`] [`append`]

stores the current conformation to a trajectory file.

Options:

- ◊ `append` : appends to previously existing file
- ◊ `write` : closes the movie file

The advantage of the trajectory file is the possibility of interpolated display as a trajectory animation. See `display trajectory` .

Example in which we create trajectory from a stack:

```
for i=1,Nof(conf)
  load conf i
  store frame
endfor
store frame write
#
display ribbon
display trajectory sstructure 20. 40.
```

ssearch

is a systematic search through torsion space combined with local minimization.

- ◊ you may globally optimize any set of energy/penalty terms including electrostatics, solvation, entropy, density correlation etc.
- ◊ you may search an arbitrary subset of variables
- ◊ you may allow full local minimization after each systematic change

◊ you may search only through centers of the preferred local `multidimensional zones` (for example `rotamers`) which is more efficient than an even grid sampling
◊ you may perform both the global search (the full [-180.,180.] range) and the local search (grid search around the current conformation).

`ssearch` [`local`] [`residue`] [*vs_Ssearch* [*vs_minimize*]] [*as_select1* [*as_select2*]]
systematically changes *vs_Ssearch* variables and carries out energy minimization with respect to the *vs_minimize* variables after each systematic conformational change. The lowest energy conformation is loaded from the conformational stack at the end of the procedure. By default every variable from *vs_Ssearch* selection goes through `nSsearchStep` evenly distributed values. The step therefore is 360 deg. over *nSsearchStep*. Option `local` imposes the grid locally around the current values of *vs_Ssearch* variables. In this case the program uses `ssearchStep` parameter.
If you want to prevent the procedure from automatically writing the stack of best conformations to a file set the `autoSavePeriod` variable to zero.

Option `residue` allows one to searche each variables of each residue independently.
See also `montecarlo` .
Example:

```
 read object "crn"  # good old crambin
 ssearch v_/14/x*   # place optimally Asn14 side-chain
 ssearch residue v_/tyr/x*   # loops through tyrosines and ssearch each separate
# ssearch residue simple vs_  # GAP model only
```

strip

`strip` *os_object* [`virtual`]
strip an ICM-molecular object from its ICM attributes and reduce it into a pdb-object. The latter are still good for graphics, superposition, basic geometric manipulations etc. Also, some chemical operations, e.g. attaching chemical groups are best performed on simpler pdb-objects. Stripping may save you a lot of memory as well.
Option `virtual` tells the command to delete the virtual atoms upon conversion. The virtual atoms (selected as `a_//vt*`) are always present in the ICM object, but are not necessary in the stripped object.
String is also used to perform operations which are not allowed for ICM object, but are allowed for simpler PDB objects (for example dragging individual atoms with a mouse) These commands include:
◊ deleting hydrogens
◊ make bond auto

Example:

```
   build smiles "c1ccccc1"
         strip a_ virtual
```

superimpose

`superimpose` [[`align` | `residue` | *ali*] [`exact`] [`minimize`]] *as_selectStatic as_selectMovable*
`superimpose` *os_static I_atomNumbers1 os_movable I_atomNumbers2*
`superimpose` *as_movableByTethers* [`reverse`]

`superimpose chemical` [`output`] | `pharmacophore` *as_selectStatic as_selectMovable*

`superimpose` *P_atompairs os_movable* # e.g. superimpose distpairs a_1.
optimally superimpose the second movable object onto the first one using selected atoms or residues as equivalent points. At least one pair of equivalent atoms needs to be provided.

Option `minimize` iteratively finds the best subset of atom pairs (see `superimpose minimize`)

Option `residue` skips residue alignment by sequence or numbers and aligns them sequentially as selected. The atoms are aligned by name. Use option `minimize`with it.

Option `reverse` in superimposition by tethers moves the 'template', rather than the selected object.

The *P_atompairs* argument allows one to superimpose by an arbitrary set of atom pairs. The atom pairs can be created with the `make distance` command or picked in GUI with the distance tool.

Selections may by of any level:

- ◊ atom selection `as_` ,
- ◊ residue selection `rs_` ,
- ◊ molecular selection `ms_`
- ◊ object selection `os_` .

Example in which we will **superimpose the selection of the binding site residues**. Perform the following steps:

- ◊ generate a master sequence alignment, e.g.

  ```
  read pdb "1ql6"
  read pdb "2phk"
  make sequence a_*.1
  Sequence(a_*.1)
  alig = Align( 1ql6_a 2phk_a )
  ```

  Edit this alignment if necessary (usually you do not need to do it)
- ◊ find the selections for the binding pocket in one or both molecules, e.g.

  ```
  bindpock = Sphere( a_2phk.atp a_2phk.a 10. )
  ```
- ◊ Align by this residues, keep the a_2phk. object where it is and change the coordinates of a_1ql6. :

  ```
  superimpoase bindpock a_1ql6.a alig
  ```

  If you do not care about the alignment, it can also be generated on the fly with the `align` option instead of the alignment name.

The second molecule can also have a selection, then the intersect of the two selections will be used for superposition.

The option defines how the two sets are aligned (the residue alignment may be explicitly provided as the *ali_* argument, and the objects are `linked` with the alignment):

`chemical` option can be used to superimpose small molecules. In this mode atom equivalence can be found either by substructure search or (if none of molecules is substructure of other) by common substructure search algorithm. Other feature of `chemical` mode is that it enumerates topologically equivalent atoms to find best superposition.

Option `output` (with option `chemical)` produces `R_2out` array with individual deviations.
alignment options:

- ◊ Default (no options): Residue alignment: by residue number. Atom alignment: by atom name for pairs of identical residues or pairs of close residues (F with Y; B with D,N; D with N; E with Qor Z, Q with Z), for other residue pairs only the backbone atoms ca,c,n,o,hn,ha are aligned.
- ◊ `align` option: *Residue* correspondence is established by *sequence alignment* using the ICM ZEGA alignment `Abagyan, Batalov, 1997` *Atom* alignment: by atom name (see the default option).
- ◊ `exact` option: Residue matching is ignored. Two atom selections are directly sequentially aligned. Numbers of atoms in two selections must coincide.
- ◊ **align exact** option: Residue alignment: `Needleman and Wunsch`. Inside residue atoms are aligned sequentially and regardless of the name.

Number of equivalent atom pairs is saved in `i_out`; resulting RMSD is saved in `r_out`; a selection of atoms in the "static" object used for superposition is saved in `as_out`, that of "movable" object in `as2_out` .

Virtual atoms. Be default, the first two virtual atoms (`vt1` and `vt2`) are automatically excluded from both selections unless the `virtual` option is explicitly specified.
Note that if the movable object is of ICM-type it is preferable to have all six `virtual` variables unfixed (e.g. `unfix V_movableObj.//?vt*`). Otherwise, if some or all of them (`V_//?vt*`) are fixed, you will get a warning, and only the partial minimization of the RMS distance possible with the given degrees of freedom will be performed.
If the explicit order of atoms is specified and two single object selections are provided, e.g.

```
superimpose a_a. a_b. {3 5 7} {10 3 5}
```

the superposition will be performed in the specified order.
The following output is produced:

- ◊ `i_out` : the total number of equivalent atom pairs superimposed (it is also equal to `Nof(as_out)`)
- ◊ `r_out` : the rms deviation for all equivalent atom pairs
- ◊ `as_out` and `as2_out` : gives the equivalent atoms in two objects.
- ◊ R_out array of 12 elements returns the superposition `transformation vector` for the `transform` command.
- ◊ with option `output` the actual deviations upon superposition will be returned in `R_2out` . This command will create table `DEV` of atomic deviations: `add columnt DEV Sarray(as_out) Sarray(as_2out) R_2out`

See also: `Rmsd`( ), `Srmsd( )`, `superimpose minimize` .

Iterative search of the best atom pair subset for superposition.

`superimpose` *as1 as2* `minimize` *options*

This procedure attempts to find the better "alignable" core in both structures **after** the atom equivalences have been established. This is important if there is a minority of atom pairs that are really different in two selections and this minority messes up the superposition and the RMSD values. Examples of that such movements include moving side-chains, loops, tails, etc.

Theory

The algorithm resembles the one published by Damm and Carlson in Biophys.J 2006,90,4558 with a few modifications, namely the adaptable st.dev. for the gaussian distribution (step 5) and the way the weighted Rmsd is calculated (in ICM it is divided by the sum of weights, rather than by n). The adaptable denominator in the distribution ensures a better quality superposition.

The ICM procedure uses the *weighted* superposition and the following procedure:

1. Start from two aligned or equivalent atom arrays **A** and **B** The atom equivalences established according to residue numbers, alignments, atom names etc. (see `superimpose` options).
2. set all weights to 1.
3. perform weighted superposition (and evaluate Rmsd, **R**).
4. Calculate the deviations $\mathbf{D_i}$ for each atom pair *i* .
5. Sort the deviations and find the deviation $\mathbf{D_x}$ corresponding to the X-quantile (the `TOOLS.superimposeMinAtomFraction` parameter). E.g. if this parameter is 0.5, you will find $D_{50,}$ the 50-percentile of the deviation array.
6. calculate the weights **W** according to following formula: $\mathbf{W_i}$ = `exp (` $\mathbf{- D^2_x / D^2_i}$) small deviations compared to this adaptable mid-scale deviation will get weights close to 1. while larger deviation will get progressively smaller weights
7. go back to step 3 unless the iteration limit `TOOLS.superimposeMaxIterations` is reached or `RMSD` is not improved any more.

This procedure will gradually find the alignable core that will cover at least X % of the pairs. The -minimize principle is also implemented in the `Rmsd` function.

To calculate RMSD values of different subsets of atoms one can use the `Srmsd` function after this molecules are superimposed. The `l_info` variable controls if the iterations are shown .

The following output is produced:

◊ `i_out` : the total number of equivalent atom pairs superimposed (it is also equal to `Nof(as_out)`)
◊ `r_out` : the weighted rms deviation for ALL equivalent atom pairs
◊ `i_2out` : the number of equivalent atom pairs that define the core for which the unweighted rms is calculated
◊ `r_2out` : the unweighted rms deviation for the 'core subset' of atom pairs deviating less than `TOOLS.superimposeMaxDeviation`
◊ `as_out` : is returned in the superimpose command and gives the atoms in the static object that have 'equivalent' counterparts in the other object. `i_2out/Real(i_out)` will give you the fraction of equivalent atom pairs in the core
◊ R_out array of 12 elements returns the superposition `transformation vector` for the `transform` command.
◊ with option `output` the actual deviations upon superposition will be returned in `R_2out` . This command will create table `DEV` of atomic deviations: `add column DEV Sarray(as_out) Sarray(as_2out) R_2out`

See also :

◊ `Rmsd(` *as1 as2* `minimize` [*option*])
◊ `Rmsd(` *as1* `tether minimize`)
◊ `Smsd(` *as1 as2 option*)

Parameters for the `minimize` option of the superposition:

◊ `TOOLS.superimposeMaxIterations`
◊ `TOOLS.superimposeMinAtomFraction`
◊ `TOOLS.superimposeMaxDeviation` determines the output of the command, namely, reports the fraction of initial set of equivalent pairs that are superimposed with distances below this limit.

sys (or unix): system command

`sys system_shell_command unix unix_shell_command`
issues a system shell command from ICM. You may use `sys` or `unix` interchangeably. However, every time your ICM script makes a system call, ICM spawns a new process. Keep in mind that some simple external operations on files and directories are possible **without** the thread-spawning `unix` command. Here is the list of what can be done without it:

command	**comment**	**unix equivalent**	**example**
`delete system` *s_file*	delete a single file	`rm` *file*	a="1crn.ob"; delete system a
`rename system` *s_f1 s_newname*	rename/move a single file	`mv` *file1 file2*	rename system "1crn.ob" "1crn_old.ob"
copy-system*s_f1 s_f2*	copy a single file	`cp` *file1 file2*	copy system "a" "b"
`set directory` *s_dirname*	change directory (cd)	`cd` *dirname*	set directory "./DOCK1"
`make directory` *s_dirname*	make a directory	`mkdir`	make directory "NEW"
`Path` (directory)	returns the path to the current directory	`pwd`	s_currDir = Path(directory)
`Sarray` (*s_filename_filter* `directory` [`all`])	returns the file list array, `all` goes to subdirectories	`ls` -1 [-R] *name_pattern*	a = Sarray("*.icb" directory)

Back to the `sys` command. By default, the ICM process waits until the system shell process has completed. `sys` must be the first word in the command. **Important**: Construction

```
 if ( <condition> ) sys system_command
```

is **illegal.** Use

```
 if (  <condition> ) then
    sys system_command
  endif
```

instead. For cross-platform compatibility, also use the following portable ICM shell variables instead of non-portable system-specific commands: `s_sysCp` , `s_sysLs` , `s_sysLtt` , `s_sysMv` , `s_sysRm`. Example:

```
 sys $s_sysLs   # cross-platform portable list command
 sys ls         # non-portable unix only ls command
```

As you might have guessed from the above example, to pass the ICM-shell variables to the *system_shell_command* one may use `integer`, `real` or `string` ICM-shell variables, protected with dollar sign ($) prefix. **Important:** passing ICM-shell variables to the UNIX command is impossible if you use an alias name (e.g. `ux)` instead of the original `unix` command.
Examples:

```
 unix grep -i myoglobin /data/pdb/brookdir.doc
 unix echo $mncalls $s_pdbDir $dielConst

 file="/data/pdb/"+Name(a_1.)      # tricky file name
 unix grep ATOM $file | wc -l      # $file will be substituted by
                                   # the value of this ICM-shell
                                   # string variable
```

See also:

◊ `Unix` function
◊ `make background` command

test

`test` *l_val* | *i_val*

This command produces an error if the condition passed to it as anrgument is not true. It is convenient for writing testing frameworks and debugging scripts.

Examples:

```
test yes
test no
test 2==2
test 2==3
```

`test real` *r_v1 r_v2*

`test exact` *I_v1 I_v2*

`test exact` *S_v1 S_v2*

`test real` *R_v1 R_v2*

`test real` *M_v1 M_v2*

`test exact` *T_v1 T_v2*

These commands test two objects to be identical. For real values, the comparison is made with a certain tolerance. Tables with advanced `parray` columns may not be properly supported.

Examples:

```
test real {2. 4.}  2.*{1. 2.}
test exact {2 4}  2*{1 2}
```

test binary

`test binary` *s_file1 s_file2*

Tests two files to be identical.

then

is one of the `ICM flow control` statements, used to perform `conditional statements`.
See also `if`, `elseif`, and `endif` .

transform

performs transformations of `3D objects` or `string arrays` in place. The geometrical transformation is defined by the `transformation vector` .

transform string arrays in place

`transform sarray` *S_array* "`tolower`"|"`toupper`"|"`trim`"

This command will transform elements of string arrays or text columns of tables in place. Three transformations are currently possible:

- ◊ "tolower"
- ◊ "toupper"
- ◊ "trim"

Example:

```
read table s_userDir + "inx/PDB.tab"
transform sarray PDB.head "tolower" # in place
```

transform molecular objects or grobs

transform molecular objects to symmetry related positions.
`transform` {*ms*|*g_grob*} *R_12transformationVector*

transform molecules (*`ms_`*) or graphics objects according to the `transformation vector`.
See also these two examples: (`example 1` and `example 2`).
You can also manually move molecules with respect to each other on the graphics screen by using the `connect` *ms_* command to choose the molecules which can be moved separately.
`transform` *ms i_transformationNumber* [`translate` [= <{`x`,`y`,`z`}>]]
transform molecules `ms_` according to the specified transformation.
i_transformationNumber is a symmetry operation number in an array of all operators of a space group. The first transformation usually keeps the object in place. The symmetry transformations are defined in a 12*n real array where each chunk of 12 real values defines 3x3 rotation matrix and translation vector {a4,a8,a12}. The complete 4x4 transformation matrix looks like this:

```
a1  a2  a3  | a4
a5  a6  a7  | a8
a9  a10 a11 | a12
------------+----
0.  0.  0.  | 1.
```

If *i_number* exceeds the number of space group symmetry transformations the symmetrical images in up to 26 surrounding cells are created. This operation is only possible, if symmetry information (sym.group name and cell dimensions) is defined for the object. Usually PDB and CSD files contain the above information, it is preserved upon `conversion`. Use the `Cell( )` or the `Symgroup( )` functions to find out if the space group is defined. If not, you may assign it to the object with the `set symmetry` `object` command. In a special case of *i_number=0*, the object is placed in the "primary" subunit of the cell (e.g. in sym.group "P 21 21 21" that is 0<x<a, 0<y<b, 0<z<c/4; currently, the *i_number=0* option is supported only for groups 1 and 19).

Option `translate` tells the command to shift the transformed coordinates back to the vicinity of the source coordinate set (`translate`) or to the vicinity of the *{x,y,z}* point provided.
Example:

```
read pdb "1sre"
copy a_1. "a1"
transform a_a1. Transform(a_a1.)[13:24]  #  Trasform with R_12transformationVec
copy a_1. "a2"
transform a_a2. 3                        # same using i_transformationNumber
```

See also `Transform`

translate

`translate` { *os* | *ms* | *g_grob* .. | `origin` } { `add` *R3_transl_vector* | *R3_destinationPoint* | *M_xyz* [`symmetry`]
translate the center of mass of the specified object(s) (*os_*) or molecule(s) (*ms_*) **to** a specified position, or, with the `add` option, **by** a *R_3translationVector* vector. If a Nx3 matrix is specified, the mean vector is calculated. You can also move molecules/objects interactively with the mouse after the `connect` command. Without the `add` option, the translation
symmetry option With the `symmetry` option the R_3translationVector should be in **fractional** coordinates. Option `add` translates **by** the specified vector from the current position. Without `add` the program tries to identify a compensating shift to a position in which the center of gravity of the selected molecule(s) has minimal positive fractional coordinates.
Examples:

```
read pdb "1fbi"
delete a_!p,q,y  # get rid of redundancies
copy a_ "a1"
translate a_a1. add symmetry {0., 0., -1.} #  shift whole object by fractional
cool a_
for i=1,10
  translate a_y add {0., 0., 0.9}  # shift molecule y  by an increment
endfor
```

To calculate a displacement vector, follow this example in which we calculate a translation vector for molecule `y` :

```
read pdb "1fbi"
delete a_!p,q,y  # get rid of reduncancies
cool a_
v1 = Rarray( Xyz( a_y/1/ca ) )
connect a_y  # now drag the molecule with the middle button and press Esc
v2 = Rarray( Xyz( a_y/1/ca ) )
vtrans = v2 - v1
```

undisplay

`undisplay` [[*ms*] `store`] *args* Opposite to `display` .

The `store` option preserves colors and representations so that they can be restored by the next display command.
Examples of the undisplay command:

```
undisplay store a_1,2         # undisplay the two molecules and memorize their
undisplay ribbon              # ribbon display not needed any more
undisplay g_icos              # a graphics object not needed any more
undisplay a_/w*,hoh*          # who cares about water molecules ...
undisplay residue labels      # just "labels" will do the same
undisplay string              # see also "delete label" command
undisplay a_//h*              # who cares about hydrogens ...
```

```
 undisplay hbond a_1./1:29      # ... and, hence, about H-bonds
 undisplay tether a_/12:20
 undisplay box
 undisplay cursor
 undisplay origin               # undisplay the coordinate frame
 undisplay volume               # deactivate the fog effect
 undisplay window
```

To get rid of the whole graphics window for fast calculations use:

```
 undisplay window                # delete GL graphics window
```

---

undisplay window

`undisplay window`

This command deletes the 3D graphics window. It may be used to speed up the calculations by avoiding the re-drawing operations. This command can also be applied from `Windows` menu of the GUI interface

See `display window`

unfix

`unfix` [`only`] *Vs_select*
unfix (set free) specified variables (such as bond lengths, angles and phases or torsions) in an ICM-object. Opposite to `fix` command. This operation can be applied to the `current object` only (use `set object` *`os_newObj`* first).
Important: since it only makes sense to unfix variables which are currently fixed, use `all variable selection` starting with capital V which selects among ALL (both free and fixed) variables, as opposed to `vs_` which selects only from FREE variables.
Examples:

```
       # only this loop has free torsions now
 unfix only V_/8:18/phi,PSI,H,M,P
```

Note that `PSI` torsion references is used for traditional residue attribution

---

wait

wait for the child ICM processes to finish, quit the child processes
`wait` [`pipe`]
allows one to synchronize multiple ICM processes spawned by the fork command.

- ◊ for the parent process: wait until all the child processes spawned with the `fork` command are finished.
- ◊ for the child processes: quit the spawned ICM process

With *pipe* option the command will synchronously prints the output from all child processes launched with `fork pipe`

See $ICMHOME/molpipe/molto3d.icm

See also: `fork` , `wait` , l_out (defines the parent), `Index`(fork [system|all]) .

---

web

web `s_url`

invokes an external `web browser` call to WWW page or local file (Html, Pdf etc). Can be used e.g. to link ICM table entries to NCBI, PDB etc. databases

Example:

```
s_ncbi= "http://www.ncbi.nlm.nih.gov/entrez/viewer.fcgi?db=protein&val="
```

```
web s_ncbi+"Q28509"
```

web table: shows an icm table with a web browser

`web` [`delete`] [*s_file*] *T* [`link` *T.S_1 s_linktype1 T.S_2 s_linktype2 ...*]
The command presents the *T_* table in your web browser window. Optional web links are interpreted according to the web link types described in the `WEBLINK.DB` array.
If the table contains chemicals, ICM creates a file with the compound images using Peter Ertl's JME classes (see also the `s_javaCodeBase` variable).

Example:

```
 read sequence "zincFing.seq"
 find prosite 1znf_m 0.3
 show SITES
 web SITES link SITES.AC "AUTO"
```

See also: `write html` , `show html`

while

`while`
is one of the `ICM flow control` statements, used to perform a `loop` in the ICM-shell calculations.
See also: `for`, `endwhile` .

write

write stuff to a disk file. Logical variable `l_confirm` defines if you'll be prompted whether to overwrite an existing file with the same name. Use option `delete` to delete (or overwrite) the existing file unconditionally.
For the list of ICM-objects you can write, and formats you can choose, see `read` and `show` commands. Generic syntax:
`write` [binary] [`append` | `delete`] { `variable` | `constant` | `expression` } *s_fileNameRoot*[.ext]
With the `binary` binary option multiple objects or classes of objects can be writtin into a single cross-platform compatible binary file. To read it use `read binary` and to read the table of its contents use `read binary list` .

Common options:

- ◊ append - appends to an existing file or creates new
- ◊ delete - overwrites an existing file

See also corresponding `read` commands.

write alignment

`write` [`alignment`] [`msf` | `fasta`] *ali_Name* [*s_fileName*] [`SEQUENCE.restoreOrigNames=yes|no`]
write alignment *ali_Name* to a file. Default extension is `.ali` . Note: if alignment is only a group of unaligned sequences, generated by the `group` command, the result will be just a `multiple sequence file`, rather than an `alignment file` (there will be no dashes at the end).
The default ICM format for an alignment looks like this:

```
#>ali sh3
# Consensus         ...#.^.YD%..+~..-#~# K~-.#~##.~~..~WW.#.   ~~.~
Fyn           ----VTLFVALYDYEARTEDDLSFHKGEKFQILNSSEGDWWEARSLTTGET
Spec          DETGKELVLALYDYQEKSPREVTMKKGDILTLLNSTNKDWWKVE--VNDRQ
Eps8          KTQPKKYAKSKYDFVARNSSELSM-KDDVLELILDDRRQWWKVR---NSGD
#Fyn              __EEEE__________________EEEEEEE____EEEEEE_____E
```

```
# Consensus   G%#P...#..#.
Fyn           GYIPSNYVAPVDSIQ
Spec          GFVPAAYVKKLD---
Eps8          GFVPNNILDIMRTPE
#Fyn          EEEGGGGEEE_____

# nID 7 Lmin 61 ID 11.5 %
```

The lines starting from hash (#) are comments and are not required
The length of each alignment block is controlled by the `sequenceLine` parameter (default value is 60). If you want to save a long alignment as one unwrapped block, increase this value (e.g. `sequenceLine=1000`)
Writing sequences in the alignment order
The sequences can be written in the alignment order with the following commands (they can be store in a little macro)

```
 macro wrSeqAli ali_ s_file ("seq.fasta")
  l_showSstructure = no
  seqname = Name(ali_) # Name returns sarray of sequence names
  for i=1,Nof(seqname)
    write sequence fasta append $seqname[i] s_file
  endfor
 endmacro
```

Resorting alignment in the order of sequence input.
Upon alignment the source sequences get reordered according to similarity. If you want to keep the original order you may use the `reorderAlignmentSeq` macro described in the `Align(` *ali_ I_newOrder*) section and then write an alignment:

```
 read sequence s_icmhome+"zincFing"
 group sequence aaa
 align aaa
 reorderAlignmentSeq aaa
 write ali_new    # reordered alignment
```

restoring the original name of the genbank sequences There is a method to swap the ICM names of sequences with the names stored in the form of the `comment` containing this text `" Orig.name: "`*other_seq_name* . If this comment exists (can be set with `set comment` *seq s*)
See also: `SEQUENCE.restoreOrigNames` , `String(` *ali_)* function.

---

write binary

`write binary` [*class1 class2 ...*] [*obj1 obj2 ...*] [*s_fileName* | `stdout`]

`write binary all` [`key=s_password`] [*s_fileName* | `stdout`] [`read only`]
write specified ICM shell objects or all objects of a classes to a single, binary, cross-platform file, or more accurately, database. The following data types are currently supported:

- ◊ `alignment`
- ◊ `distance` # pairdistances, like hbonds etc.
- ◊ `grob`
- ◊ `iarray`
- ◊ `image`
- ◊ `integer`
- ◊ `logical`
- ◊ `map`
- ◊ `matrix`
- ◊ model # prediction and classification models
- ◊ `object`
- ◊ page
- ◊ `preference`
- ◊ `rarray`
- ◊ `real`
- ◊ `sarray`
- ◊ `sequence`

◊ `slide`
◊ `string`
◊ `table`
◊ tree

The catalogue of the database can be obtained with the `list binary` command. The default file name is `"icm.icb"`, and the default extension is `.icb` (stands for ICm Binary file). The system objects or the objects with property

Options:

```
* --all save all objects in the shell (system variables are skipped)
* --key= ~~s_password protect the file with a password. To open this file with t
From the command line: to open a protected file, use
 read binary [all]  [edit]
* --read --only  : saves a file in a read only mode for other users.
```

Examples:

```
ii = {2 3 4}
rr = {2. 3.4 5.5}
g = Grob("CELL",{1. 1. 1.})
g2 = g*2. # twice as large
write binary iarray rarray grob   # the default file is icm.icb
 Info> 4 icm shell objects icm.icb

list binary   # looks at "icm.icb"
   1 ii                              iarray             20
   2 rr                              rarray             32
   3 g                               grob             1788
   4 g2                              grob             1788

delete ii
read binary name={"ii"}
  Info> 1 icm shell objects read from icm.icb

write binary grob "aaa"
  Info> 2 icm shell objects aaa.icb
```

See also: `list binary`, `read binary`

---

write iarray

`write` [`iarray`] *I_name* [*s_fileName*]

write rarray

`write` [`rarray`] *R_name* [*s_fileName*]

write sarray

`write` [`sarray`] *S_name* [*s_fileName*]

write matrix

`write` [`matrix`] *M_name* [*s_fileName*]
write an array or a matrix to a disk file. Default file extensions are `.iar`, `.rar`, `.sar`, or `.mat`, respectively.
See also: `read iarray`, `read rarray`, `read sarray`, `read matrix`.

---

write molcart

`write molcart` [`mol` | `separator=`*s_sep* [`header`]] `table=`*s_dbtable* *s_filename* [*connection_options*]

Exports database table *s_dbtable* in `SDF` or CSV/TSV file format (with or without `header)`. If the format is not specified explicitly, it is guessed from the *s_filename* extension.

The Molcart connection may be specified by `connection_options`.

See also: `molcart`, `make molcart`

write several arrays

`write` [{ `column` | `database`] } `array1` `array2` .... [*s_fileName*]
write arrays in the `column` or `database` format to a disk file. Default file extension is `.db`
See also: `read database`.

writing tethers

If you imposed tethers between you current object and another object and you want to quit the session and then restore you setup, you can use the following trick:

```
# first let us create an object a_ly6. tethered to template a_x.
read alignment s_icmhome+"sx"
read pdb s_icmhome+"x"
build model ly6 a_x.m   # a new object a_ly6. created and tethered
#
write string String( a_//T ) "tTz.str" # tethered model atoms
write string String( a_//Z ) "xTz.str" # x-template atoms
write object a_x,ly6. "tx.ob"
#
quit
#
% icm
read object "tx.ob"
read string "tTz.str" name="tTz"
read string "xTz.str" name="xTz"
set tether $xTz $tTz exact    # tethers restored
```

write table

writing ICM table in text format `write` *T_table1* [*T_table2* ..] [`field=` *s_delimiter*] [*s_fileName*]
write the *T_table* `table` to a disk file `*.tab`. It will have two header lines with table name and field name information, followed by the values.
The default extension `.tab` is appended automatically. The ICM text table format has a header which allows one to read this table back to icm with the `read table` command
Example:

```
 group table t {1 2 3} "a" {"one","two","three"} "b"
 t1=t[2:3]
 write t t1 "tt"  # write both tables in one file
 delete table     # read both tables
```

writing tables in CSV or TSV formats
`write` *T_table1* [`header`] [`separator=` *s_delimiter*] [*s_fileName*]
if the separator or the `s_fieldDelimiter` variable contain just a simple symbol (e.g. comma or tab), ICM will write a comma-separated or tab-separated table with the first line containing the field names, e.g.

```
 group table t {1 2 3} "a" {"one","two","three"} "b"
 write t header separator="," "t.csv"
 unix cat t.csv
 a,b
 1,one
 2,two
 3,three

 write t separator="," "t.csv"  # without header
 unix cat t.csv
 1,one
 2,two
 3,three
```

To `read a table` in comma-separated format with the headers, use the following commands:

```
read table separator="," header name="t" "t.csv"
```

writing tables in a binary format
`write binary` *T_table1 T_table2 .. s_file*

`write binary tables` *s_file*
The most compact and fast format is the binary format. Any object can be saved to and read from a binary project file with `".icb"` (ICM-binary) extension.
See also `write database T` and `write column`.

Writing/exporting an sdf/mol file

`write table mol` *s_sdfFileName* [`index`] [`compress`]
writes an ICM chemical spreadsheet as a `mol/sdf` file. All the property columns are added as feature records to individual mol-entries. Options:

- ◊ `index` adds sequential order number as an additional property named `IX` (it may be useful as an ID).
- ◊ `compress` skips 9 columns for each atom field, and unused bond fields in the output .sdf file

Example:

```
read table mol "ex_mol.mol" name="t" unique
write table mol t
```

write column

`write column array1 array2` .... [*s_fileName*] [`separator=` *s_Separators*]
write arrays in a multi- `column` format to a disk file.
Examples:

```
 read column s_icmhome + "res.tab"    # amino acid properties
 write column aa flexInd  "tm.tab"    # two columns
```

If you want to write all the entries of an ICM-table you may do the following.
Examples:

```
 read column s_icmhome + "res.tab" # a set of isolated arrays
 group table RES $s_out  # create an ICM-table RES (s_out : array names)
 write RES               # write in the 'table' layout
 write database RES      # write table RES in the 'database' layout
```

Default file extension is `.col`.
See also: `read column`, `show column`. `read table`, `show table`.

write database

`write database` [`html`] { `array1 array2` .... | `table` } [*s_fileNameRoot*]
write several arrays or a table in a database format to a file (usually tables are written in a multi `column` format). This command can also be used to save a subset of arrays of a table in a specific order. Option `html` writes the table with appropriate HTML tags. See also `read database` `write table`, `show database`.
Example:

```
 resnam = {"ala" "glu" "arg"}
 reschg = { 0., -1., 1.}
 write database resnam reschg "a" # default extension ".db" will be added
#
 group table t resnam reschg
 write database t.reschg t.resnam "a"  # reverse the order</tt>
```

write drestraint

`write drestraint` [*as*] [*s_fileNameRoot*]
write distance restraints of the current object to a file.
See also: `drestraints` and `drestraint types`.

write drestraint type

`write drestraint types`
write `drestraint types` to a file. You may define your own types with the `set drestraint type` command or by editing a *.cnt file.

write factor

`write` [`factor`] `factor_Name` [*s_factorFileNameRoot*]
writes crystallographic structure factors to a file.

write gamess

`write gamess` [`charge`|`energy`|`cartesian`] [`memory=`*i_Mb*] [`store=`*i_intsize*] [`fix=`*vs*] [`type="DFT"`] [`new`] *as*

See also:

- ◊ `gamess`
- ◊ `read gamess`

write grob

Commands for exporting `graphical objects`.
`write grob off` *g_name* [*s_fileName*]

Export in Object File Format (OFF). This is a simple file format supporting points, faces (triangles), edges (lines), normals, per-vertex colors. The default extension is ".off".

`write grob wavefront` *g_name* [*s_fileName*]

Export in Wavefront OBJ/MTL file format. Usually the file will be exported in many files. The object geometry and structure (points, faces, lines, groups of points) are stored in an ".obj" file. Coloring (material) properties are stored in a separate ".mtl" file. Material textures are exported in the image format in which they are stored, usually JPEG or PNG.
`write` {`grob` | *g_name*} [*s_fileName*] [`append`]
Write/append to a disk file. If *g_name* is not specified, all `grob`s are written. Depending on object features, they may be exported in OFF or Wavefront OBJ file formats.

See also: `write image`, `write postscript`, `read grob`.

write html

`write html` *T s_outputHtmlFileName* [`link` *T.S_1 s_linktype1* ...][`split=` *n*] [`none`]
writes the *T_* table with HTML tags to a file. Interpret web links according to the web link types described in the `WEBLINK.DB` array. If the table contains chemicals, ICM creates a file with the compound images using Peter Ertl's JME classes (see also the `s_javaCodeBase` variable).

Arguments and Options

- ◊ `link` *table_column1 s_link_type1* ... ...

◊ split : divide a table with large number of rows into tables *n* rows each to speed up the rendition.
◊ none : suppress Molsoft Logo.

Example:

```
read sequence "zincFing.seq"
find prosite 1znf_m 0.3
show SITES
write SITES "tmp.htm" link SITES.AC "AUTO"
web SITES link SITES.AC "AUTO"
```

See also:

◊ show html,
◊ web T_
◊ write string *s_htmlText s_file*

---

write image

write image [{ png | targa | gif | rgb }] [display] [print] [postscript [{ print | preview }]] [compress] [stereo] [{ color | bw }] [window= *I_xyPixelSizes*] [store] [*s_fileName*] write the current screen image to a file. The default image file format is tif . The png-format is the most compact and is recommended for web-publishing. The default settings are stored in the IMAGE table. Some of them can be overridden by the following options:

◊ display - allows one to view the saved image or postscript image file. The viewer is defined by the s_imageViewer variable for targa, gif, rgb and tif images and by the s_psViewer variable for the postscript images.
◊ postscript - write Adobe postscript-bitmap file rather than TIFF-file. See also write postscript command which generates **vectorized** scalable high quality postscript files.
◊ preview - add low-resolution preview to postscript file for some EPS-compliant image viewers (i.e. Irix showcase®). Resolution, and therefore the size, of the added preview is defined by the IMAGE.previewResolution (default 10).
◊ print - print the postscript file. It will not work for non-postscript images, in which case you may use the display option and print from your image viewing program instead.
◊ compress - use packbits lossless compression standard for **.tif** files. Compression of this kind is currently a standard feature of all baseline TIFF-reading programs. Compression is a standard feature of the .gif and .png formats.
◊ stereo - generate stereo image even from the mono display. Tiff-files preserve the image screen dimensions for each image in a stereo-pair. Stereo-base for postscript files is controlled by the IMAGE.stereoBase parameter and equals 2.35" (60mm) by default.
◊ store - generates an internal image in ICM album (see also store image)
◊ color or bw - color or black-and-white options surpass IMAGE.color logical variable.
◊ window= *I_xyPixelSizes* - generate image of any arbitrarily large resolution (e.g. window=3*View(window) to triple the resolution). Suppose that you want to make a poster of 4613 by 2888 pixels. This resolution is not achievable on a 1200x1024 screen. The image area will be divided into many squares and the program will merge them into one image of large resolution. This option will not work with string labels. Example:

```
nice "1crn"  # resize the image
delete label
IMAGE.compress = no  #just a plain uncompressed image
write image window={4000,2700} # for slides

write image window=2*View(window) # double the res.
```

IMAGE.generateAlpha logical variable controls if the alpha channel information is added to the SGI rgb and tif image files. This additional channel describes opacity of the image pixels and makes the background transparent. Images generated with alpha channel can be nicely superimposed in the IRIX showcase since their backgrounds are

transparent.
Examples:

```
 display a_1crn. ribbon
 write image "a"              # a.tif image - about 1400 kB
 write image "p" compress     # p.tif image - about 88 kB
 write image postscript stereo display "aaa.eps"
 write image 2*View(window)   # hi-res, may screw up labels
 unix lp -c a.eps             # print if you like the result
```

See also: `write grob`, `write png` - a different version of the png writer: does not allow arbitrary resolution, but allows transparent background, `write postscript`.

write 2D image

`write image` *image-array* [*S_filenames*|*s_directory_to_save*|*s_single_file_name*]

save images stored in ICM into the specified location.

Example:

```
nice "1crn"
# make 3 images with default names and add them to the default album 'album'
make image
make image
make image
write image album[1] "myimage.png"
write image album[1:2] {"img1.png","img2.png"} #specify names to be used
write image album s_tempDir #save all images into the s_tempDir
```

write 2D chemical image

`write image` [*chemical*|*chemArray*] [*s_fileName*] [`window` = { *i_width i_height* }] [`display` = *s_chemViewString*] [`IMAGE.bondLength2D` = *r_bondLengthInch*] [`IMAGE.lineWidth2D` = *r_lineWidth*] [`transparent`] [`sstructure`=*s_smarts*]

write chemical depiction to a file. File extension defines image type. If multiple chemicals are provided, separate file will be created for each one.

You can increase resolution by adjusting IMAGE.bondLength2D and/or *window* argument.

Chemical view options can be adjusted by providing *display* argument. See `set property chemical view` for format description.

Use *transparent* option to generate transparent background.

The `display` option: Each character in *s_chemViewString* codes single chemical view option.

- ◊ "H" : Hetero-atom hydrogens
- ◊ "T" : Terminal hydrogens
- ◊ "S" : Atom stereo labels
- ◊ "X" : Do not show explicit hydrogens
- ◊ "A" : Aromatic rings"
- ◊ "C" : Show 'chiral/racemic' flag
- ◊ "3" : Do not show 3D as 2D
- ◊ "U" : Unique atom classes
- ◊ "N" : Atom numbers
- ◊ "M" : Monochrome atom labels
- ◊ "W" : Don't show atom text labels. Colors half of the atom's adjustment bond with the element color (Like wire in 3D)
- ◊ "R" : Don't show atom text labels. Draw color square instead.

Example:

```
write image Chemical("CCO") "ethanol.png" IMAGE.bondLength2D = 0.8
write image Chemical({"C1CCN(CC1)c1ccccc1", "CCN(C)c1ccccc1" }) display="AR" # a
```

write alignment image

`write image` *alig* [*s_fileName*] [`i_resIncrease=2`] `delete`

write alignment image to a file. File extension defines image type.

You can increase resolution by providing integer argument.

You can set alignment view property either manually in GUI or using `set property alignment` command.

Example: (export all alignments in high resolution with profile enabled)

```
S_al = Name( alignment )
for i=1,Nof(S_al)
  s_al = S_al[i]
  set property $s_al 2048   # turn on the profile
  write image $s_al Name(s_al) + ".png" 4 delete  # write high-res (x4) png imag
endfor
```

---

write index

General text and specialized content (e.g. write index mol) index files.

General text parsing `write index` *s_inputFile* `pattern=`*s_startPattern* [`add=`*s_endPattern*] [*s_outIndexFile*]

general indexing of a text file, Example in which .sdf files are index as text (compare with `write index mol`)

```
 write index "/tmp/huge.sdf" pattern="" add="$$$$" # file huge.inx will be savec
#  write index mol "/tmp/huge.sdf"  # another method that will create an entry-b
```

See also: `read index`, `read index table`, `Sarray index`

Specialized index files

`write index` [`mol` | `mol2` | `fasta` | `swiss` | `mmcif`] *s_inputFile* [*s_outIndexFile*]

`write index` [`swiss` | `mol` | `mol2` | `fasta`] *T_dbDescription* [*s_outIndexFile*]
calculate and write index for a database file described by the control table *T_dbDescription*, or by the *s_inputFile* in the short form of this command.

Output

- ◊ the index file
- ◊ `i_out` contains the number of entries indexed

Simple example:

```
write index mol "/data/nci.sdf" "nci.inx" # creates nci.inx file
show i_out
read index "nci" name="x"  # creates internal index table x
Path(x)  # returns /data/nci/
read table x[1:100] # load first 100 molecules to ICM
```

The *T_dbDescription* table, optional for `mol/sdf` and `mol2/ml2` files, contains information about the database file (files) and fields to be indexed. It may have the following components in the header:

- ◊ DIR - string directory name
- ◊ FI - sarray of database files
- ◊ EXT - extension of the database files

After the header there is a string array containing the list of fields. To create this table either define it in a file or use the `group table` command. All text fields (except data) are hashed for fast searching.

The `fasta` option allows one to index the NCBI non-redundant databases.

See also: `makeIndexChemDb` macro to do indexing in one step, `mol`, `mol2` .
Example:

```
 write index mol "drugs.sdf" # the index file is saved to the current directory
 read index "drugs"
 write index mol "./drugs.sdf"

 group table t {"ID","DE","KW","SQ"} "fd" header "/data/swissprot/" \
       "DIR" {"sprot"} "FI" ".dat" "EXT"
               # we created control table t
 write index swiss t "/data/icm/inx/SWISS.inx"  # make index and save to a file
 read index "/data/icm/inx/SWISS.inx"           # read index
 show SWISS[2:5]
 show SWISS.ID=={"12AH_CLOS4","1431_LYCES","B3AT_CHICK"}
 read sequence SWISS.DE=="DNA-BINDING"
```

See also: `Path` (*T_indexTable*), write-index-mmcif

write index blast

`write index sequence` *s_blastRootFileName*
create a set of blast-formatted binary files for searches with the `find database` command. The command will use all the sequences currently loaded into the ICM-shell and will create the following compact binary files (the first three files are the same as those generated by the `setdb` blast command):

- ◊ *name.psq* binary sequences
- ◊ *name.pin* pointers/index
- ◊ *name.phd* sequence headers
- ◊ *name.psa* # optional: relative solvent accessibilities for each residue.

The relative solvent areas file is saved only if the sequence was generated from an object in which the areas had been calculated with the `show area surface` command. If the `.psa` file is present, ICM will modulate the scores with the accessibilities (it will be more permissive for the accessible residues).

If you want to do the opposite (i.e. given the three or four blast files, generate one fasta sequence file), use the
`find database write` *s_DBpath* `output=` *s_fastaFile*
command.
Simple example (indexing can also be done with the blast setdb routine):

```
# copy to the current directory and edit the icm.cfg file
# make sure that MnSequences is larger than the number of
# sequences in your database
#
 read sequence "fak.seq" # fasta formatted
 write index sequence "/tmp/db1"
 delete sequences
 a=Sequence("MERTDITMKH KLGGGQYGEV YEGVWKKYSL TVAVKTLKED TMEVEEFLKE")
 find database a "/tmp/db1" 0.001
```

A more direct way of making the blast files is via the `formatdb` utility, e.g.

```
formatdb -i /data/blast/dbf/FASTA/pdbaa -n /tmp/p_db
./icm
read sequence swiss web "10KD_VIGUN"
find database fast=10 10KD_VIGUN "/tmp/p_db"
```

See also:

- ◊ `write index fasta` *s_file.fasta s_file.inx*

write library

`write library` [`append`] [`auto`] *as_entryAtom* [`exit=` *as_exitAtom*] *s_libFileRoot*
save a selected molecule, residue or a fragment as an `ICM-library entry`. Use `set charge`, `set bond type` and, possibly, `build hydrogens` before writing an entry. We recommend you to do this operation in an interactive session: display your molecule and `Ctrl-Click` the first and last atoms if needed. There are two different

situations:

read sequence "aaa.seq"

1. the molecule/residue/fragment does not belong to an ICM-type object. For example, you have a pdb-file with a new molecule you would like to create an ICM-library entry from. In this case do NOT use option `auto` and note that the resulting entry will only be a draft, since energy parameters of atoms (`atom codes` plus related types of `van der Waals`, `hydrogen bondings` `solvation`), as well as parameters of `torsions`, `bond angles`, `phase angles`, and `bond lengths` will have to be further manually adjusted. Enter the command and you will be prompted for the first and the last atoms of the entry. The purpose of this procedure is to create a regular ICM-tree, create extra bonds if there are cycles and give atoms unique names. Some additional editing of the entry may be required to correct fixed and free torsions suggested by the program. To declare a certain variable free, enter '+' in the appropriate field.
2. the molecule/residue/fragment belongs to an ICM-type object. In this case you may use option `auto` since all the information is there already. The program only needs to extract the molecular subtree according to the specified selection.

Example:

```
 build string "nter glu cooh" # build glutamic acid residue
 strip   # convert it to a non-ICM object
 write library a_def./2/ha "./tm" name ="new" auto  # reroot it
# Now the entry atom is a_//hg2
 LIBRARY.res = LIBRARY.res // "./tm"
 build string "new" # read the rerooted residue
 display
```

---

write map

`write` `m_map` [*s_fileName*]
write specified map to a binary file with specified file.
`write` { `map` | `m_map1 m_map2` ... }
write all maps or specified maps to corresponding files (the names for the files are generated from map names, the `m_` prefix is removed from the file names).
`write` `xplor` `m_map` ... [*s_fileName*]
write the specified map to a Xplor-formatted file.
Example:

```
 make map potential "ge,gc" Box(a_)
 m_gc... done
 Info> Map m_gc created. GridStep=0.50 Dimensions: 16 11 17, Size=2992
 m_ge... done
 Info> Map m_ge created. GridStep=0.50 Dimensions: 16 11 17, Size=2992
 write m_ge m_gc
 Info> 1 map written to file  ge.map
 Info> 1 map written to file  gc.map
```

---

write model: update or create the loop database file

`write` `model` [`append`] *s_lpsFile*
writes a compressed representation of the protein structure to the specified loop file (`"def.lps"` by default). To create a large database, read the object list and write a loop over all objects, e.g.

```
# prepare pdbUniq list and ..
 read sarray "pdbUniq.li"
 for i=1,Nof(pdbUniq)
   read object s_xpdbDir+pdbUniq[i]
# add further filters
   write model append "icm.lps"
   delete object
 endfor
```

To make the program use this file , redefine the `LIBRARY.lps` file name to, say `"./icm.lps"`

write mol

`write mol` [`exact`] *as_select* [*s_fileName*]
write selected atoms in the `mol` -file format. By default the formal charges (see the `set charge` command) are saved. If the selection contains multiple objects, each `object` will be treated as a separate `mol` entry in an `.sdf` file. (e.g. `write mol a_*.H "tmp.sdf"` Multiple molecules inside each object will be included as parts of one `mol` entry.

Options

- ◊ `exact`: preserve the ICM-atom names (like c1, c2).
- ◊ `charge`: write the MCHG section containing the atomic real charges.

See also `read mol "file.sdf"`, `show mol "file"`.

write mol2

`write mol2` [`exact`] [`formal`] *as_select* [*s_fileName*]
write selected atoms in the `mol2` -file format (extension .ml2). Options:

- ◊ `exact` preserves the ICM-atom names (like c1, c2).
- ◊ `formal` writes formal atomic charges instead of the real ones. Adds USER_CHARGES (XXXXXX) tag to the header

See also `read mol2 "file"`, `show mol2 "file"`.

write movie

`write movie` *s_file* [`on` [`exact`]] [*video_options*]

- create a movie file. Open it for writing.

Available *video_options:*

- ◊ `size=` *r_bitsPerSecPerPixel* (default = 4.). There is a tradeoff between file size and movie quality. Larger number means high quality and large files.
- ◊ `frame=` *i_framesPerSecond* (default = 25)
- ◊ `group=` *i_gop* (default = 100) . GOP stands for 'Groups Of Pictures' that is a group used for compression
- ◊ `name=` *s_title* . The movie title.
- ◊ `comment=` *s_comment*
- ◊ `set=` *s_codecflags*
- ◊ `heavy` - use best video recording quality possible

Some useful related shell variables:

`MOVIE.quality` (real)	the default number used for the 'size' parameter
`MOVIE.qualityAuto` (logical)	lets the engine to increase the video quality for movies produced in smaller resolution

When the `on` option is specified this command also starts frame grabbing (see below), so that one **write movie** command may be used instead of two.

`write movie on [exact]`

- start frame grabbing.

Frame grabbing is a video recording mode which allows the user to create movies in interactive mode. The `exact` option specifies when the frames are saved

- ◊ no option: frames are saved every 25 msec. This mode allows one to record ICM session activity in real-time.
- ◊ with `exact` option frames are saved every time the view is updated (the *frame-based* timing). This mode is more useful when used in scripts, as it is possible to control updates (see e. g. `display`) from an ICM script. The

frame-based timing generates nicer movies when the computer is not fast enough for real-time grabbing.

Update-based frame grabbing works correctly with time-based ICM features, such as `rocking/rotation`, `smooth slide transitions`, `display trajectory`, `display stack`. When the frame grabbing is enabled, these commands slow down the graphics updates if necessary to provide movie frame grabbing at the requested frame rate (e. g. 25 frames per second).

`write movie off`

- stop frame grabbing

`write movie frame` [`smooth`] [*nframes*=1] [`antialias`] [`background`|`transparent=`*r*]

- save *nframes* individual frames

`smooth` is a very powerful option allowing to create blending effects. It writes *nframes* to make a smooth transition from the previous frame. Each frame is an interpolation between the previous and current frame. If the option `smooth` is used when writing the first movie frame, **fade-in** effect is created, i. e. the command writes blended frames transforming empty scene into the current picture.

Option `background` may be used in combination with `smooth` to create a **fade-out** effect from the last frame to empty background. In general, `write movie frame background` writes an empty scene frame.

Option `antialias` applies full-scene anti-aliasing, which improves the video quality. In GUI also consider 'high quality' button and shadows (in combination with option `exact`)

Option `transparent` allows one to create frames which are blended with the background to create fade-in/fade-out effects.

`write movie exit`

- stop recording and close the file.

Example with smooth transition effects:

```
read pdb "1crn"
display
write movie "ForCannes.mov"
display wire
write movie frame 5
display ribbon
write movie frame 45 smooth
for i=1,100
  rotate view Rot({0. 1. 0.} , -1.)
  write movie frame
endfor
undisplay
write movie frame 50 smooth
write movie exit
```

Example with still image, fade-in and fade-out effects.

```
read pdb "1ekg"
display a_
color background lightblue
write movie "ItCameFromTheSky.avi"
write movie frame smooth 25 # fade-in (25 frames is one second)
write movie frame 25 antialias # still image
write movie frame smooth  background 25 # fade-out
write movie exit
```

Example featuring rotation and a more complicated way of creating fade-in/fade-out effects:

```
read pdb "1ekg"
display a_
write movie "Vertigo.mov"
```

```
for i=1,50
  write movie frame transparent=(51-i)/50. # fade-in
endfor
for i=1,100
  rotate view Rot({0. 1. 0.} , -1.)
  write movie frame # write rotated image
endfor
for i=1,50
  write movie frame transparent=i/50. # fade-out
endfor
write movie exit
```

write object

`write object` [`options`] [*as_selection*] [*s_fileName* [`rename`]]
write an ICM molecular object (or many selected ICM-objects) in binary `ICM format` to a file. A single object can be renamed in the file according to the *s_fileName*, if option `rename` is specified. **Important**: only **whole** ICM object may be written by this command, and file extension will always be `.ob`.
Options (defaults shown in bold):

- ◊ `append` : append to a multiple-object file
- ◊ `rename` : rename the single object to *s_fileName* (leave out path and extension) .
- ◊ `short` : write a compressed file for non-ICM objects without b-bactors and occupancies.
- ◊ `strip` : write a `strip`ped object (i.e. drop information about variables and rigid bodies present in an object of the ICM type).
- ◊ `auto={yes| no}` : if `yes` the program automatically identifies which atom requisites to save. For example, if molecule is displayed, the view will be saved with the object. Properties such as occupancy and charge are considered essential if the values are not identical for all the atoms. If `auto=no`, the `OBJECT` table controls the output.
- ◊ `occupancy={yes|no}` : occupancy field
- ◊ `charge` ={ yes|no} : partial atomic charges
- ◊ `bfactor` ={ yes|no} : b-factors
- ◊ `display` ={yes| no} : the current view of your molecular object(s), including `graphics planes` The written display attributes are automatically restored upon reading of the object.
- ◊ library={yes|**no**} : currently not used.

See also: `read object`, `write pdb`, `OBJECT`, `strip` .
Example:

```
 read object s_icmhome+"crn.ob"
 build string "se ala his" name="AH" # second object named "AH"
 write object a_2. "alahis" rename   # rename obj. to "alahis"
 display a_1./1:40 ribbon            # display and save with graphics attributes
 display a_1./12 cpk
 display a_2. xstick
 write object a_*. "twoobj" display=yes # both objects in one file
 write object a_1. append "twoobj"      # yet another object
```

write object simple

`write object simple` [*as_selection*] [*s_fileName*]
write a compressed object. The information preserved in the compressed description of the object is limited to 3 coordinates and certain atom names (non-protein atom names will not be preserved and reduced to just one character) plus all residue and molecule requisites. For a PDB-type file, a simple object is the most compact for store and fastest to read. They are used in the compact fold library.

write object (parray)

`write object` *objParray s_file*

writes object parray into .ob file. This file can be read either with `read object` or `read object parray` commands.

write pdb

`write` pdb [`exact`] [`charge`] [`nosort`][*as_selection*] [*s_fileName*]
write a molecular (sub)object in `PDB` format. Normally atoms of each amino acid are sorted in the following order:

```
ATOM     19  N   GLN O   3      -4.565   0.000  -4.592  1.00 20.00
ATOM     20  CA  GLN O   3      -4.712   0.000  -6.037  1.00 20.00
ATOM     21  C   GLN O   3      -6.194   0.000  -6.420  1.00 20.00
ATOM     22  O   GLN O   3      -7.063   0.000  -5.549  1.00 20.00
<i>the rest</i>
```

Also the n-terminal nitrogen and its hydrogens are assigned to the first amino acid. Options are the following:

- ◊ `charge` saves atomic charges instead of occupancies and atomic radii instead of B-factors;
- ◊ `exact` keeps the names of hydrogen atoms the same as in ICM objects (i.e. the first character is 'h'). Without this option names of hydrogen atoms are transformed like this:

```
 h11 &rarr; 1H1
 h12 &rarr; 2H1
```

- ◊ `nosort` retain the original ICM order of atoms

Default file extension is `.pdb`.
See also: `write object`, `read pdb`.

write png

`write` png [`transparent`] [`window=` *I_xyPixelSizes*] [*s_fileName*]

this is a new version of the png writer (`write image` png). This version supports option `transparent` that makes the background transparent. Options:

- ◊ `transparent` sets alpha to max value for all pixels with background color. Without this option the alpha values are set to 0.
- ◊ `window` = { Width, Height } in pixels. If you want to specify just one size and determine the second from the aspect ratio, use zero, e.g. `window={0,600}` to set height to 600 pixels
- ◊ *s_fileName* self-evident

write postscript

`write postscript` [`display`] [`stereo`] [`preview`] [{ `color` | `bw` | `dash`]} [*i_quality*] [*r_gammaCorrection*] [*s_filename*]
create **vectorized** postscript model of the screen image. Instead of the bitmap snapshot this command generates lines, solid triangles and text strings corresponding to the displayed objects. Since the postscript language is directly interpreted by high-end printers, the printed image may be even higher quality than the displayed image. The final resolution is limited only by the printer since the original image is not pixelized. Warning: there may be inevitable side-effects for some types of solid images at the intersection lines of solid surfaces (i.e. large scale `cpk` representation, hint: use `display skin` instead).
The default settings are stored in the `IMAGE` table. Some of them can be overridden by the following options and arguments:

- ◊ `reverse` - makes white background in the saved postscript file.
- ◊ `display` - allows one to view the saved postscript file. The viewer is defined by the `s_psViewer` variable.
- ◊ `stereo` - generate stereo image even from the mono display. Stereo-base is controlled by the `IMAGE.stereoBase` parameter and is 2.35" (6cm) by default.
- ◊ `preview` - generates postscript preview according to the `IMAGE.previewer` command string and the `IMAGE.previewResolution` parameter.
- ◊ `color` or `bw` - color or black-and-white options surpass `IMAGE.color` logical variable.

◊ dash - is a great variant of the black-and-white option to generate lines of different width and style. The line colors of your screen image are interpreted according to the following table:
· gold - double solid black line
· pink - triple solid black line
· magenta - dash1
· orange - dash2
· brown - dotted line
· the rest - solid black line
Examples:

```
 read object s_icmhome+"crn.ob"
 display a_crn.  # display wire model of crambin
 color a_//ca,c,n pink         # triple width backbone
 color a_/arg/!ca,c,n magenta  # dashed lys side chains
# zoom your picture to fill the whole graphics window
 write postscript dash stereo display
```

◊ *i_quality* (default=3, possible range: 1:100) - defines a parameter in a smoothing procedure. Each side of an elementary triangle is divided into *i_quality* sections and color of all the *i_quality*^{2} smaller triangles is calculated to yield smooth transitions. Optimal value of the parameter depends on an image. Only large scale images may require *i_quality* values above 10. Only in an extreme case of a single triangle on a page with red, blue and green vertexes, one may need *i_quality* of 100.

◊ *r_gammaCorrection* allows one to lighten or darken the image by changing the *gamma* parameter. A gamma value that is greater than 1.0 will lighten printed picture, while a gamma value that is less that 1.0 will darken it. You may adjust your gamma correction parameter for your printer with respect to your display and add this setting to the _startup file.

Examples:

```
 read object s_icmhome+"crn.ob"
 display a_crn. brown skin   # molecular surface
                              # Hugh wants to have a look
 write postscript 1 1. "divine_brown" display
                              # change parameters for the printer
 write postscript 5 2. "divine_brown"
                              # and print it
 unix lp -c divine_brown.eps
```

See also: write image, write grob.

write pov

write pov [image] [*r_aspectRatio*] [*s_fileName*]
writes a pov-ray object file which can be processed with the pov-ray ray-tracing program.
Example:

```
 buildpep "ala his trp"
 display cpk
 make grob image
 write pov "x"
% pov-ray x.pov
```

write sequence

write { sequence | *seq* } [{ fasta | swiss | pir | gcg | msf }] [*s_fileName*]
write all sequences or the specified sequence *seq_* to a file in one of specified formats. The default format is the fasta format.

write session

`write` `session` [*s_fileName*]
write commands from an ICM session to a file. Default file name is "_session.icm". This is a simple text file with icm commands. Feel free to edit the file
Example:

```
 ..
 a=1
 history 10
 write session
  Info> 4 history lines written to file  _session.icm
```

See also: `history` and `delete session` commands.

write stack

`write` `stack` [`simple`] [*s_fileName*]
write the current state of the `conformational stack` to a disk file. Starting from May, 2003, version ICM3.022, the stack file is compressed by default. The stack file is not compressed if the `simple` option is used. Default file extension is `.cnf`.
See also: `show stack`, `delete stack`, `read stack`, `read conf`.

write system preference

`write` `system` `preference` [*preferenceName*]

saves the *persistent* user preferences to a operating system specific location (~/.config/Molsoft.conf on Unix, plist file on Mac, registry on Windows, see `preference system` for details). Note that only the registed persistent preferences can be saved this way, any other parameters, new or existing need to be changed in a user_startup.icm script or directly in a command or macro.

This command tracks if a preferences has been changed The command without additional arguments will save ALL CHANGED preferences. Examples:

```
 write system preference  # save modified preferences
#
 TOOLS.edsDir = "/data/eds/"
 write system preference TOOLS.edsDir  # save only this preference
```

write vs_var

`write` [*vs_variables*][*s_fileName*]
write a variable selection `vs_` to a disk file.
Default file extension is `.var` .
See also: `read variable`.

Functions

ICM-shell functions are an important part of the ICM-shell environment. They have the following general format: *FunctionName* (*arg1*, *arg2*, ...) and return an ICM-shell object of one of the following types: `integer`, `real`, `string`, `logical`, `iarray`, `rarray`, `sarray`, `matrix`, `sequence`, `profile`, `alignments`, `maps`, `graphics objects, a.k.a. grob` and `selections`.
The order of the function arguments is fixed in contrast to that of `commands`. The same function may perform different operations and return ICM-shell constants of different type depending on the arguments types and order. ICM-shell objects returned by functions have no names, they may be parts of algebraic expressions and should be formally considered as 'constants'. Individual 'constants' or expressions can be assigned to a named variable. Function names always start with a capital letter. Example:

```
 show Mean(Random(1.,3.,10))
```

Abs

absolute value function.
`Abs` (*real*) - returns `real` absolute value.
`Abs` (*integer*) - returns `integer` absolute value.
`Abs` (*rarray*) - returns `rarray` of absolute values.
`Abs` (*iarray*) - returns `iarray` of absolute values.

`Abs` (*map*) - returns `map` of absolute values of the source map.

Examples:

```
a=Abs(-5.)                  # a=5.
print Abs({-2.,0.1,-3.})    # prints rarray {2., 0.1, 3.}
if (Abs({-3, 1})=={3 1}) print "ok"
```

Acc

accessibility selection function. It returns residues or atoms with relative solvent accessible area greater than certain threshold. **Important:** The surface area must be calculated before this function call. The `Acc` function just uses surface values, it does not reevaluate them. Therefore, make sure that the `show area` command (or `show energy`, `minimize` , etc. with the `"sf"` surface term turned on), has been executed before you use the `Acc` function. If you specify the threshold explicitly, it must range from 0.0 to 1.0, otherwise it is set to 0.25 for residue selections and 0.1 for atom selections.
`Acc` (*rs* , [*r_Threshold*])
- returns `residue selection`, containing a subset of specified residues `*rs_* for which the ratio of their current accessible surface to the standard exposed surface is greater than the specified or default threshold (0.25 by default). ICM stores the table of standard residue accessibilities in an unfolded state calculated in the extended `Gly-X-Gly` dipeptide for all amino acid residue types. It can be displayed by the `show residue type` command, or by calling function `Area(` *s_residueName* ), and the numbers may be modified in the `icm.res` file.
The actual solvent accessible surface, calculated by a fast dot-surface algorithm, is divided by the standard one and the residue gets selected if it is greater than the specified or default threshold. (*r_Threshold* parameter is 0.25 by default).
`Acc` (*as_select*, [*r_Threshold*])
- returns `atom selection`, containing atoms with accessible surface divided by the total surface of the atomic sphere in a standard covalent environment greater than the specified or default threshold (0.1). Accessibility at this level does not make as much sense as at the residue level. The standard surface of the atom was determined for standard amino-acid residues. Note that hydrogens were **NOT** considered in this calculation. Therefore, to assign surface areas to the atoms use
`show surface area a_//!h* a_//!h*`
command or the
`show energy "sf"`
command.
You may later propagate the accessible atomic layer by applying `Sphere(` *as_* , 1.1), where 1.1 is larger than a typical X-H distance but smaller than the distance between two heavy atoms. (the optimal *r_Threshold* at the atomic level used as the default is 0.1, note that it is different from the previous).
Examples:

```
                                  # let us select interface residues
read object s_icmhome+"complex"
                                  # display all surface residues
show surface area
display Acc( a_/* )
                                  # now let us show the interface residues
display a_1,2
color a_1 yellow
color a_2 blue
show surface area a_1 a_1         # calculate surface of
                                  # the first molecule only
```

```
                              # select interface residues
                              # of the first molecule
color red Sphere(a_2/* a_1/* 4.) & Acc(a_1/*)

read object s_icmhome+"crn"
show energy "sf"
display
display cpk Acc(a_//* 0.1)   # display accessible atoms

show surface area            # prior to invoking Acc function
                             # surface area should be calculated
color Acc(a_/*) red          # color residues with relative
                             # accessibility > 25% red
```

Acos

arccosine trigonometric function Returns angles in degrees.
`Acos` (`real` | `integer`) - returns the `real` arccosine of its real or integer argument.
`Acos` (*rarray*) - returns the `rarray` of arccosines of *rarray* elements.
Examples:

```
print Acos(1.)               # equal to 0.
print Acos(1)                # the same

print Acos({-1., 0., 1.})    # returns {180. 90. 0.}
```

Acosh

inverse hyperbolic cosine function.
`Acosh` (`real` | `integer`) - returns the `real` inverse hyperbolic cosine of its real or integer argument.
`Acosh` (*rarray*) - returns the `rarray` of inverse hyperbolic cosines of *rarray* elements.
Examples:

```
print Acosh(1.)               # returns 0
print Acosh(1)                # the same

print Acosh({1., 10., 100.})  # returns {0., 2.993223, 5.298292}
```

Align

family of the alignment functions. These function return an `alignment` icm-shell object and perform

- ◊ sequence alignment (with the `Needleman and Wunsch` algorithm with zero gap end penalties (`ZEGA`),
- ◊ structural alignment, or
- ◊ sub-alignment extraction

Pairwise sequence alignment or sequence-structure alignment

`Align` ([*sequence1*, *sequence2*] [`area`] [`M_scores`])

- returns `ZEGA`- `alignment`. If no arguments are given, the function aligns the first two sequences in the sequence list. For sequence alignments, the ZEGA-statistics of structural significance (`Abagyan, Batalov, 1997`) is given and can be additionally evaluated with the `Probability` function. The reported pP value is -Log(Probability,10).

Returned variables:

- ◊ `i_out` - the number of identical residues in the alignment
- ◊ `r_out` - contains `Log` (*Probability_of_structural_dissimilarity*) only for pairwise alignments

◊ r_2out - percent identity of the alignment.

Simple pairwise sequence alignment
Align( )
Align(*seq1 seq2*) - returns an alignment. The alignMethod preference allows you to perform two types of pairwise sequence alignments: "ZEGA" and "H-align". If you skip the arguments, the first two sequence are aligned.
Example:

```
   read sequences s_icmhome+"sh3.seq" # read 3 sequences
   print Align(Fyn,Spec)              # align two of them
   Align( )                           # the first two
   a=Align( sequence[1] sequence[3] ) # 1st and 3rd
   if(r_out > 5.) print "Sequences are struct. related"
```

Aligning DNA or RNA sequences Make sure to read the dna.comp comp_matrix before using the Align function, e.g.

```
a=Sequence("GAGTGAGGG GAGCAGTTGG CTGAAGATGG TCCCCGCCGA GGGACCGGTG GGCGACGGCG")
b=Sequence("GCATGCGGA GTGAGGGGAG CAGTTGGGAA CAGATGGTCC CCGCCGAGGG ACCGGTGGG")
read comp_matrix s_icmhome+"dna.cmp"
c = Align(a,b)
```

Aligning with custom residue weights or weights according to surface accessible area
Align(*seq1 seq2* area)
Option area will use relative residue accessibilities to weight the residue-residue substitution values in the course of the alignment (see also accFunction).
The weights must be positive and less than 2.37 . Try to be around or less than 1. since relative accessibilities are always in [0.,1.] range. Values larger than 2.37 do not work well anyway with the existing alignment matrices and gap parameters. Use the Trim function to adjust the values, e.g. Trim(myweights , 0.1,2.3)).
E.g.

```
 read pdb "1lbd"
 show surface area
 make sequence
  Info> sequence  1lbd_m  extracted
 1lbd_a                       # see the relative areas
 read pdb sequence "1fm6.a/"  # does not have areas
 Info> 1 sequence 1fm6_a  read from /data/pdb/fm/pdb1fm6.ent.Z
 ali3d = Align( 1lbd_a 1fm6_a area )
```

This can also be used to assign *custom* weights with the following commands

```
 set area seq1 R_weights  # must be > 0. and less than 2.37
 Align( seq1 seq2 area )
```

Introducing positional restraints into the alignment matrix
Align(*seq1 seq2 M_positionalScores*)
If sequence similarity is in the "twilight zone" and the alignment is not obvious, the regular comp_matrix{residue substitution matrix} is not sufficient to produce a correct alignment and additional help is needed. This help may come in a form of the positional information, e.g. histidine 55 in the first sequence must align with histidine 36 in the second sequence, or the predicted alpha-helix in the first sequence preferably aligns with alpha-helix in the second one.
In this case you can prepare a matrix of extra scores for each pair of positions in two sequences, e.g.

```
 seq1 = Sequence("WEARSLTTGETGYIPSA")
 seq2 = Sequence("WKVEVNDRQGFVPAAY")
 Align()
 # Consensus   W.#.  .~~.~G%#P^
           WEARSLTTGETGYIPS--
           WKVE--VNDRQGFVPAAY
 m = Matrix(17,16,0.)
 m[10,4] = 3. # reward alignment of E in seq1[10]  and E in seq2[4]
 Align(seq1 seq2 m )
 # Consensus   W.#      E    ~G%#P^
           WEARSLTTGE----TGYIPS--
           WKV------EVNDRQGFVPAAY
```

The `alignSS` macro shows a more elaborate example in which extra scores are prepared to encourage alignments of the same secondary structure elements.
Warning. The alignment procedure is rather subtle and may be sensitive to the gap parameters and the comparison matrix. Avoid matrix values comparable with gap opening penalty.

See also: `Probability(` *ali* ..) for local alignment reliability.

Local pairwise structural alignment

Two types of structural alignments or mixed sequence/structural alignments can be performed with the `Align` function.
`Align(` *seq_1 seq_2* `distance` [*i_window*] [*r_seq_weight*]) - performs local structural alignment, using **distance RMSD** as structural fitness criterion. The RMSD is calculated in a window *i_window* (default 10) and the dynamic programming algorithm then subtracts the window averaged local sequence alignment score multipled by the *r_seq_weight* >= 0. The sequence weight can be any positive number or zero.
`Align(` *seq_1 seq_2* `superimpose` [*i_window*] [*r_seq_weight*]) - performs local structural alignment, using **superposition followed by coordinate RMSD** calculation as structural fitness criterion. The RMSD is calculated in a window *i_window* and the dynamic programming algorithm subtracts the local sequence alignment scores multipled by *r_seq_weight* .
In both cases the function uses the dynamic algorithm to find the alignment of the locally structurally similar backbone conformations.
The alignment based on optimal *structural superposition* of two 3D structures may be different from purely sequence alignment
Preconditions:

- ◊ sequences must be `link`ed to 3D molecules to access the coordinate information;
- ◊ two 3D structures must have superposable subsets

The residue-label-carrying atoms (see the `set label a_` command) will be used for structural superpositions. *r_seq_weight* is used to add sequence amino acid substitution values to the 3D similarity signal.

See also: `align` *ms1 ms2* function

Deriving an alignment from tethers between two 3D objects
`Align` (*ms*) - returns `alignment` between sequences of the specified molecule and the template molecule to which it is tethered. The alignment is deduced from the tethers imposed.

Example:

```
build string "se ala his leu gly trp ala" name="a"  # obj. a
build string "se his val gly trp gly ala" name="b"  # obj. b
set tether a_2./1:3 a_1./2:4 align                 # impose tethers
show Align(a_2.1)         # derive alignment from tethers
write Align(a_2.1) "aa"   # save it to a file
```

Extracting pairwise alignment sequences from a multiple alignment

`Align` (*ali*, *seq_1*, *seq_2*) - returns a pairwise sub- `alignment` of the input alignment *ali_*, reorders of sequences in the alignment according to the order of arguments.
Extracting a multiple alignment of a subset of sequences from a multiple alignment
`Align` (*ali*, *I_seqNumbers*) - returns a reordered and/or partial `alignment` . Sequences are taken in the order specified in *I_seqNumbers.*

Examples:

```
            # 14 sequences
read alignment msf s_icmhome + "azurins"
            # extract a pairwise alignment by names
aa = Align(azurins,Azu2_Metj,Azur_Alcde)
```

```
             # reordered sub-alignment extracted by numbers
 bb = Align(azurins,{2 5 3 4 10 11 12})
```

Resorting alignment in the order of sequence input with the `Align` (ali_, I_seqNumbers) function.
Load the following macro and apply it to your alignment. Example:

```
 macro reorderAlignmentSeq( ali_ )
  nn=Name(ali_)  # names in the alignment order
  ii=Iarray(Nof(nn))
  j=0
  for i=1,Nof(sequence)  # the original order
   ipos = Index( nn, Name(sequence[i] ) )
   if ipos >0 then
     j=j+1
     ii[j] = ipos
   endif
  endfor
  ali_new = Align( ali_ ii )
  keep ali_new
 endmacro
```

Angle

a family of functions calculating planar angles. The most detailed is Angle (table) (see below). They calculates planar angle in degrees.
`Angle` (*as_atom*) - returns the planar angle defined by the specified atom and two previous atoms in the ICM-tree. For example, Angle(a_/5/c) is defined by C-Ca-N atoms of the 5-th residue. You may type:

```
 print Angle(  # and then click the atom of interest.
```

`Angle` (*as_atom1* , *as_atom2* , *as_atom3*) - returns the planar angle defined by three atoms.
`Angle` (*R_3point1* , *R_3point2* , *R_3point3*) - returns the planar angle defined by the three points.
`Angle` (*R_vector1* , *R_vector2*) - returns the planar angle between the two vectors.

`Angle` (*as* `table`) - returns a table of all covalently bound atom triplets with their two bond lengths and a planer angle. Example:

```
read pdb "1xbb"
t=Angle(a_H table)
sort t.angle
show t
```

`Angle` (*as|rs|ms|os as_filter* `error`) - returns a `rarray` of minimal angles within each specific unit of the selection. The size of the array depends on the *level* of the selection. Used to detect errors (too small angles).
Examples:

```
 d=Angle( a_/4/c )                          # d equals N-Ca-C angle
 print Angle( a_/4/ca a_/5/ca a_/6/ca )     # virtual Ca-Ca-Ca planar angle
```

The rotation angle corresponding to a transformation vector is returned as `r_out` by the `Axis(` R_12) function.

Area

calculates surface area. A quick guide:

`Area(` *grob* `[error]`)   `r`

`Area(` *as* | *rs*)   *R_atomAreas|R_resAreas* `# needs surface calculation beforehand`

`Area(` *rs* `type`)   *R_maxAreas_in_GLY_X_GLY*

Area(*as R_typeEyPerArea* energy) *R_atomEnergies*

Area(*seq*) *R_relAreasPerResidue*

Area(*s_icmResType*) r

Area(*rs rs_2*) *M_contactAreas*

Area(*rs rs_2* distance [*min*(4.) *max*(8.) [*Ca_Cb_len*(2.3)]])
M_0_to_1_contact_strength

Note that if an atom selection is provided as an argument the surface area needs to be computed beforehand with the show area or show energy "sf" command. The detailed description can be found below:

Area (*grob* [error]) - returns real surface area of a solid graphics object. Option error makes it return the fraction of the surface that is not closed to detect the holes or missing patches in what supposed to be a closed surface. (e.g.

```
g = Grob("SPHERE",1.,2)
show Area(g)
if(Area(g error)>0.01) print "Surface not closed" # check for holes
```

See also: the Volume(*grob*) function, the split command and How to display and characterize protein cavities section.
Area (*as* [[*R_userSolvationDensities*] [energy]]) - returns rarray of pre-calculated solvent accessible areas or energies for selected atoms `*as_* . This areas are set by the show area surface|skin of show energy "sf" commands. Make sure to clean up the areas with the set area a_//* 0. command before computing the areas with show energy command since the command ignores hydrogens.

With option energy returns the product of the individual atomic accessibilities by the atomic surface energy density. The values of the density depend on the surfaceMethod preference and are stored in the icm.hdt file. The "contant tension" value of the preference is a trivial case in which all areas are multiplied by the surfaceTension parameter. For the "atomic solvation" and "apolar" styles, the densities depend on atom types. Normally the atomic solvation densities are taken from the icm.hdt file where the density values are listed for each hydration atom type for "atomic solvation" and "apolar" styles. However, you can provide your own array of *n* values *R_userSolvationDensities* with the number of elements less or equal to the number of types to overwrite the first *n* types.

Examples:

```
read object s_icmhome+"crn.ob"
set area a_//* 0.
surfaceMethod = "apolar"
show energy "sf" # only heavy atoms
Area( a_/15:30/* ) # areas of this atoms
#
# Now let us redefine the first three solvation parameters
# of icm.hdt and calculated E*A contributions of selected atoms
#
Area( a_/15:30/* {10., 20. 30.} energy)
```

Area (*rs*) - returns rarray of pre-calculated solvent accessible areas for selected residues `*rs_* . These accessibilities depend on conformation.
Area (*rs* type) - returns rarray of maximal standard solvent accessible areas for selected residues `*rs_* . These accessibilities are calculated for each residue in standard extended conformation surrounded by Gly residues. Those accessibilities depend only on the sequence of the selected residues and do NOT depend on its conformation. To calculate normalized accessibilities, divide Area (*rs_*) by Area (*rs_* type)
Example:

```
read object s_icmhome+"crn.ob"
show surface area
a=Area(a_/* )      # absolute conformation dependent residue accessibilities
```

```
b=Area(a_/* type ) # maximal residue accessibilities in the extended conformatio
c = a/b            # relative (normalized) accessibilities
```

`Area` (*resCode*) *r_standard_area*

- returns the `real` value of solvent accessible area for the specified residue type in the standard "exposed" conformation surrounded by the Gly residues, e.g. `Area("ala")`. It is the same value as the `Area` (.. type) function.
`Area(` *seq*) *R_relAreasPerResidue*

- returns an array of relative areas per residue stored with the sequence by the `make sequence` command from molecules in which the areas had been computed beforehand. Note that the sequence keeps only a very limited accuracy areas. Example:

```
read pdb "1crn"
show area surface
make sequence  # 1crn_a now has relative areas
group table t Sarray( a_/* residue) Area(1crn_a)  Area(a_/*)/Area(a_/* type)
show t
```

Important : "pre-calculated" above means that before invoking this function, you should calculate the surface by `show area surface` , `show area skin` or `show energy "sf"` commands.
Examples:

```
 build string "se ala his leu gly trp lys ala"
 show area surface       # calculate surface area
 a = Area(a_//o*)        # individual accessibilities of oxygens

 stdarea = Area("lys")   # standard accessibility of lysine

# More curious example
 read object s_icmhome+"crn.ob"
 show energy "sf"        # calculate the surface energy contribution
                         # (hence, the accessibilities are
                         # also calculated)

 assign sstructure a_/* "_"
                         # remove current secondary structure assignment
                         # for tube representation
 display ribbon
                         # calculate smoothed relative accessibilities
                         # and color tube representation accordingly
 color ribbon a_/* Smooth(Area(a_/*)/Area(a_/* type) 5)
                         # plot residue accessibility profile
 plot Count(1 Nof(a_/*)) Smooth(Area(a_/*)/Area(a_/* type) 5) display
```

See also: `Acc(`) function.

Area contact matrix

(also see the simplified distance-based contacts strength calculation below)
`Area` (*rs_1 rs_2*) - returns `rarray` of **areas of contact** between selected residues. You can do it for intra-molecular residue contacts, in which case both selections should be the same, i.e. Area(a_1/* a_1/*) ; or, alternatively, you can analyze intermolecular residue contacts, for example, Area(a_1/A a_2/A). See also the `Cad` function, and example in `plot area` in which a contact matrix is calculated via interatomic Ca-Ca distances. The table of the pairwise contact area differences is written to the `s_out` string which can later be read into a proper table via: `read column group name="aa" input=s_out` and sorted by the area (see below).
Example:

```
 read object s_icmhome+"crn.ob"  # good old crambin
 s=String(Sequence(a_/A))
 PLOT.rainbowStyle="blue/rainbow/red"
 plot area Area(a_/A, a_/A) comment=s//s color={-50.,50.} \
    link transparent={0., 2.} ds

 read object s_icmhome+"complex"
 plot area Area(a_1/A, a_2/A) grid color={-50.,50.} \
    link transparent={0., 2.} ds
```

`Area(` *rs rs_2* `distance` [*min*(`4.`) *max*(`8.`) [*Ca_Cb_len*(`2.3`)]])
M_0_to_1_contact_strength

- evaluates the strength of residue contact based on the projected and extended Ca-Cb vector. It works with both converted and unconverted objects and needs ca, c, and n atoms for its calculation only to be independent on the presense of Gly residues.

By default the procedure finds a point about 1.5 times beyond Cb along the Ca-Cb vector (2.3A) and calculates the distance matrix between those point. Then the distances are converted into the contact strength:

- ◊ 0. for distances larger than *max_distance* (default 8. A)
- ◊ 1. for distances smalle than *min_distance* (default 4. A)
- ◊ (*max- dist*)/(*max* - *min*) for distances between *max* and *min*

All three parameters can be redefined, e.g.

```
read pdb "1crn"
m = Area( a_/A a_/A distance 4. 7. 2.5 )
```

This matrix can also be used to evaluate the contact difference between contacts of two proteins, e.g.

```
read pdb "1crn"
read pdb "1cbn"
make sequence a_*.A
aln=Align(1crn_a 1cbn_a)
m1=Area( a_1crn.a/!Cg a_1crn.a/!Cg distance ) # !Cg excludes non-matching gappe
m2=Area( a_1cbn.a/!Cg a_1cbn.a/!Cg distance )
diff = Sum(Sum(Abs(m1-m2)))/Sum(Sum(Max(m1,m2)))
simi = 1.-diff
printf "  Info> dist=%.2f similarity=%.2f or %1f%\n" diff simi,100.*simi
```

Asin

arcsine trigonometric function Returned values are in degrees.
`Asin` (`real` | `integer`)
- returns the `real` arcsine of its real or integer argument.
`Asin` (*rarray*)
- returns the `rarray` of arcsines of *rarray* elements.
Examples:

```
print Asin(1.)              # equal to 90 degrees
print Asin(1)               # the same

print Asin({-1., 0., 1.})   # returns {-90., 0., 90.}
```

Asinh

inverse hyperbolic sine function.
`Asinh` (`real`)
- returns the `real` inverse hyperbolic sine of its real argument.
`Asinh` (`rarray`)
- returns the `rarray` of inverse hyperbolic sines of *rarray* elements.
Examples:

```
print Asinh(1.)               # returns 0.881374
print Asinh(1)                # the same

print Asinh({-1., 0., 1.})    # returns {-0.881374, 0., 0.881374}
```

Ask

interactive input function. Convenient in macros.
`Ask(` *s_prompt*, *i_default*)
- returns entered `integer` or default.
`Ask` (*s_prompt*, *r_default*)

- returns entered `real` or default.
`Ask` (*s_prompt*, *l_default*)
- returns entered `logical` or default.
`Ask` (*s_prompt*, *s_default* [`simple`])
- returns entered `string` or default. Option `simple` suppressed interpretation of the input and makes quotation marks unnecessary by automatically adding quotes around your input text.
Examples:

```
 windowSize=Ask("Enter window size",windowSize)
 s_mask=Ask("Enter alignment mask","xxx----xxx")

 grobName=Ask("Enter grob name","xxx")
 display $grobName

 show Ask("Enter string, it will be interpreted by ICM:", "")
          #e.g. Consensus( myAlignm )

 show Ask("Enter string:", "As Is",simple)
          #your input taken directly as a string
```

See also: `Askg`

Askg

interactive input function that generates a GUI dialog. **Return entered text** `Askg(` *s_prompt*, *i_default*) *s_returnsTheInputString*

E.g.

```
Askg( "Enter your name", ""  ) # empty default
Askg( "Enter your name", "Michael"  )
```

Return the pressed button. `Askg(` *s_Question*, `"Reply1/Reply2/.."` `simple`) *s_theReply*

Makes a GUI dialog with the question and several alternatives separated by a slash. This dialog returns one of the string selected ,e.g. `"Yes"`, `"No"` , or `"Cancel"` for the "Yes/No/Cancel" argument. Example:

```
s = Askg("Do you like bananas?","Yes/No/Fried only",simple)
if s=="Fried only" print "Impressive"
```

Creating a special chemical dialog for library enumeration. This one is very specialized and is used in combi-chem generator.

`Askg(` *chem_scaffold* , `enumerate`) *s_makeLib_React_Args*

`Askg(` *chem_reaction* , `enumerate`) *s_makeLib_React_Args*

prompts for arguments for the `enumerate library` or `make reaction` commands to create a combinatorial library. To use this function you need to have the chemical array objects with Markush-scaffolds or reactions, plus the building blocks loaded into ICM. The function returns a `string` with the agruments for the `enumerate library` or `make reaction` commands. E.g.

```
args = Askg( scaff1 enumerate )
enumerate library scaff1 $args
```

`Askg(` *s_dialogDeclaration*) `"yes"/"no"`

Generates a dialog from `GUI dialog description text`. Values from each input field can be accessed either by :

$*field_num*

or

Getarg(*i_field_num* gui)

```
buf  = "#dialog{\"Select InSilco Models\"}\n"
buf += "#1 l_Passive_GUT_Absorption (yes)\n"
buf += "#2 l_ToxCheck (no)\n"
buf += "#3 l_hERG_QSAR (yes)\n"
buf += "#4 s_Comment_Here ()\n"

Askg(buf)

print $1, $2, Getarg( 3 gui ), $4
```

Using Askg in shell, html-docs and table tool panels. These variants of the Askg function can also be used as a part of an ICM script in dialogs generated from built-in html `documents`, or in actions associated with `tables`.

See also : `gui programming`

Atan

arctangent trigonometric function Returned values are in degrees.
`Atan` (`real` | `integer`)
- returns the `real` arctangent of its real or integer argument.
`Atan` (*rarray*)
- returns the `rarray` of arctangents of *rarray* elements.
Examples:

```
print Atan(1.)                     # equal to 45.
print Atan(1)                      # the same.

print Atan({-1., 0., 1.})          # returns {-45., 0., 45.}
```

Atan2

arctangent trigonometric function. Returned values are in degrees.
`Atan2` (*r_x*, *r_y*)
- returns the `real` arctangent of *r_y/r_x* in the range -180. to 180. degrees using the signs of both arguments to determine the quadrant of the returned value.
`Atan2` (*R_x R_y*)
- returns the `rarray` of arctangents of *R_y/R_x* elements as described above.
Examples:

```
print Atan2(1.,-1.)                         # equal to 135.
print Atan2({-1., 0., 1.},{-0.3, 1., 0.3}) # returns phases {-106.7 0. 73.3}
```

Atanh

inverse hyperbolic tangent function.
`Atanh` (*real*)
- returns the `real` inverse hyperbolic tangent of its real argument.
`Atanh` (*rarray*)
- returns the `rarray` of inverse hyperbolic tangents of *rarray* elements.
Examples:

```
print Atanh(0.)                     # returns 0.
print Atanh(1.)                     # returns error

print Atanh({-0.9999, 0., .9999})  # returns { -4.951719, 0., -4.951719 }
```

Atom

transforms the input selection to atomic level or returns an atom level selection. Function is necessary since some of the commands/functions require a specific level of selection.

`Atom`(*as*|*rs*|*ms*|*os*)   *as_atomLevelSel* - a selection level transformation function

Atom(*vs*) *as_firstAtomMovedByVar* - each variable be it a bond length, bond angle, torsion angle or phase angle in the ICM tree has a single atom that is first moved when this variable is changed. This function returns this first atom(s).

Atom(*as_icmAtom i*) # i-th preceding atom - this function also uses the concept of the ICM tree and returns atoms *i* - th links before the selected one.

Atom(*as1* [*as_where*] symmetry) - returns a selection of atoms that are topologically equivalent to one atom defined by *as1* . The optional second selection argument *as_where* allows one to *narrow* down the search for the equivalent atoms to the specified selection.

```
build smiles "C1CCCC1"  # a cyclopentane
Atom( a_//c2 symmetry ) # returns 4 other equivalent carbons, c1,c3,c4,c5
#
build string "AFA"  # a tripeptide with phenylalanine
Atom( a_/3/ce1 a_/3 symmetry )  # returns ce2 in phe
```

Atom(*as* tether) - returns a sub-selection of *as* that has tethers .

Atom(*vs i*) # i-th preceding atom for variables

Atom(*label3d* [*i_item*]) *as*

Atom(*pairDist_or_hbondPairDist*) *as* make distance or make bond commands can be used to create distance lines and labels or hbonds, respectively, in the format of a "distance" object; The Atom function then will return the atoms referenced in the object. E.g. display Atom(hbondpairs) xstick cpk
Examples:

```
 asel=Acc(a_2/his)             # select accessible His residues of
                               # the second molecule
 show Atom(asel)               # show atoms of these residues
 show Atom( v_//phi )          # carbonyl Cs
```

See also: the Res, Mol, and Obj functions.

Augment

creates augmented affine 4x4 space transformation matrix or adds 4th column to the coordinate matrix.
Augment(*R_12transformationVector*)
- rearranges the transformation vector into an augmented affine 4x4 space transformation matrix .
The augmented matrix can be presented as

```
 a1  a2  a3  | a4
 a5  a6  a7  | a8
 a9  a10 a11 | a12
 ------------+----
 0.  0.  0.  | 1.
```

where {a1,a2,...a12} is the *R_12transformationVector* . This matrix is convenient to use because it combines rotation and translation. To find the inverse transformation simply inverse the matrix:

```
 M_inv = Power(Augment(R_12direct),-1))
 R_12inv = Vector(M_inv)
```

To convert a 4x4 matrix back to a 12-transformation vector, use the Vector(M_4x4) function.

See also: Vector (the inverse function), symmetry transformations, and transformation vector.
Augment (*R_6Cell*)
- returns 4x4 matrix of oblique transformation from fractional coordinates to absolute coordinates for given cell parameters {*a b c alpha beta gamma*}.
This matrix can be used to generate real coordinates. It also contains vectors A, B and C. See also an example.

Example:

```
 read object s_icmhome+"crn.ob"
 display a_crn.          # load and display crambin: P21 group
 obl = Augment(Cell( ))  # extract oblique matrix
 A = obl[1:3,1]          # vectors A, B, C
 B = obl[1:3,2]
 C = obl[1:3,3]
 g1=Grob("cell",Cell( )) # first cell
 g2=g1+ (-A)             # second cell
 display g1 g2
```

`Augment(` *R_3Vector* `)` - appends 1. to a 3D vector *x,y,z* (resulting in *x,y,z,1.*) to allow direct arithmetics with augmented 4x4 space transformation matrices.
`Augment(` *M_XYZblock* `)` - adds `1.,1.,..1.` column to the *Nx3* matrix of with *x,y,z* coordinates to allow direct arithmetics with augmented 4x4 space transformation matrices.

`Augment(` *M_3x3_rotation R_3trans* `)` - adds `0.,0.,0.,1.` row the *3x3* rotation matrix . Then it adds the translation vector as the first three elements of the 4th column.

Axis

calculates rotation/screw axis corresponding to a transformation
`Axis(` { *M_33Rot* | *R_12transformation* } `)`
- returns `rarray` with *x*,y,z components of the normalized rotation/screw axis vector. Additional information calculated and returned by the function:

◊ `r_out` rotation angle (in degrees);
◊ `r_2out` helix rise;
◊ `R_out` 3-rarray with a middle point on the axis.

See also: `How to find and display rotation/screw transformation axis`

Blob

`Blob(` *s_text* ['`hex`'|'`base64`'] `)`

Creates `blob` from `string`. Hex or Base64 conversion is applied if specified.

`Blob(` *any_variable* `binary )`

Serialize any shell variable into `blob`

`Blob(` *blob_serialized* `read )`

Un-serialize `blob` into shell variable.

Example:

```
read pdb "1crn"
convert auto
make map potential
c = Collection( )
c["ob"] = a_          # store object
c["map"] = m_atoms    # store map
s_base64 = String( Blob( c binary ) 'base64') # serialize collection into base64
                                              # now it can be passed between CGI

delete a_*.
delete m_atoms c
c = Blob( Blob( s_base64 'base64' ) read )    # convert s_base64 to blob and un-
load object c["ob"]
m_atoms = c["map"]
display a_
display m_atoms
```

Bfactor

crystallographic temperature factors or custom atom parameters.
`Bfactor` ([*as* | *rs*] [`simple`]) - returns `rarray` of b-factors for the specified selection of atoms or residues. If selection of `residue level` is given, the average residue b-factors are returned. B-factors can also be shown with the command `show pdb`.
Option `simple` returns a normalized b-factor. This option is possible for X-ray objects containing b-factor information. The `read pdb` command calculates the average B-factor for all non-water atoms. The normalized B-factor is calculated as ***(b-b_av)/b_av*** . This is preferable for coloring ribbons by B-factor since these numbers only depend on the ratios to the average. We recommend to use the following commands to color by b-factor:

```
color ribbon a_/ Trim(Bfactor( a_/ simple ),-0.5,3.)//-0.5//3. # or
color a_// Trim(Bfactor( a_// simple ),-0.5,3.)//-0.5//3.  # for atoms
```

This scheme will give you a full sense of how bad a particular part of the structure is.
See also: `set bfactor`.
Examples:

```
read pdb "1crn"
avB=Min(Bfactor(a_//ca))      # minimal B-factor of Ca-atoms
show Bfactor(a_//!h*)         # array of B-factors of heavy atoms
color a_//* Bfactor(a_//*)    # color previously displayed atoms
                              # according to their B-factor
color ribbon a_/A Bfactor(a_/A) # color the whole residue by mean B-fac.
```

Boltzmann

returns the `real` Boltzmann constant = 0.001987 kcal/deg.
Example:

```
deltaE = Boltzmann*temperature  # energy
```

Box

the 3D graphics box function. This box can be displayed with the `display box` command or by left-double-clicking on a `grob`, and interactively moved and resized with the mouse. One can select atoms inside a box by this operation: `as_ & Box( )`
`Box` ([`display`]) - returns the 6- `rarray` with ***{X_{min} ,Y_{min} ,Z_{min} ,X_{max} ,Y_{max} ,Z_{max} }*** parameters of the graphics box as defined on the screen. With the `display` keyword, the function returns {0. 0. 0. 0. 0. 0.} if the box is not displayed (by default it returns the last 6 values).
`Box` (`center`) - returns the 6- `rarray` with *Xcenter,Ycenter,Zcenter,Xsize,Ysize,Zsize* parameters of the graphics box as defined on the screen.
`Box` (*as* [*r_margin*]) - returns the 6- `rarray` with *Xmin,Ymin,Zmin,Xmax,Ymax,Zmax* parameters of the box surrounding the selected atoms. The boundaries are expanded by *r_margin* (default: `0.0`).

Examples:

```
build string "se ala his"  # a peptide
display box Box(a_/2 1.2)  # surround the a_/2 by a box with 1.2A margin
color a_//* & Box( )
```

`Box` ({ *g* | *m* | *R_6box* } [*r_margin*])
- returns the 6- `rarray` with *Xmin,Ymin,Zmin,Xmax,Ymax,Zmax* parameters of the box surrounding the selected grob or map. The boundaries are expanded by *r_margin* (default: `0.0`).

Bracket

bracket the grid potential map by value or by space.
`Bracket` (`m_grid` [*r_vmin r_vmax*])
- returns the truncated `map` . The map will be truncated by value. The values beyond *r_vmin* and *r_vmax* will be set to *r_vmin* and *r_vmax* respectively.
`Bracket` (`m_grid` [*R_6box*])
- returns the modified `map` . All the values beyond the specified box will be set to zero.
Example:

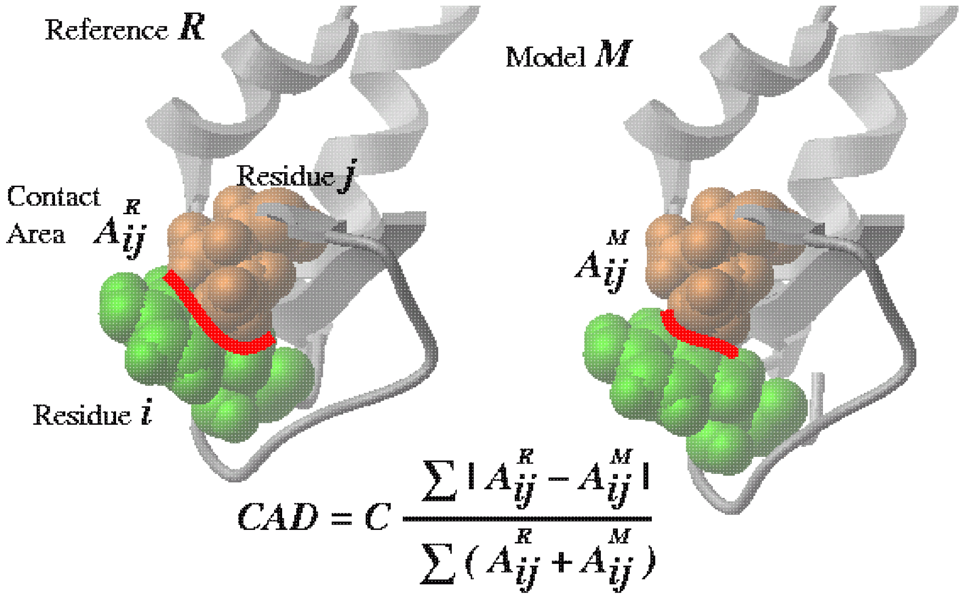
```
make map potential "gh,gc,gb,ge,gs" a_1 Box()
m_ge = Bracket(m_ge, Box( a_1/15:18,33:47 )) # redefine m_ge
```

See also: `Rmsd`(map) and `Mean`(map), `Min`(map), `Max`(map) functions.

Cad

Contact Area Difference function to measure geometrical difference between two different conformations of the same molecule. Cad, as opposed to `Rmsd`, is contact based and can measure the difference in a wide range of model accuracies. Roughly speaking it measures the surface weighted fraction of native contacts. Can be used to evaluate the differences between several NMR models, the accuracy of models by homology and the accuracy of docking solutions.
Cad can measure the geometrical difference between two conformations in several different ways:

◊ between two conformations of the **same** protein based on full atom residue-residue contact area calculation, `Cad`(..)
◊ between two conformations of the same protein based on Cbeta-Cbeta distance evaluation (`Cad1{Cad}(.. `distance`) .ICM uses an empirically derived *ContactStrength(Cb-distance*) function.
◊ between two homologous structures based preservation of the residue contacts through the alignment (`Cad` (.. `alignment`)) . The contact strength in this case is also derived from the inter-residue distances.



$$CAD = C\,\frac{\sum |A_{ij}^{R} - A_{ij}^{M}|}{\sum (A_{ij}^{R} + A_{ij}^{M})}$$

Comparing two conformations of the same molecule via residue-residue contact conservation.

`Cad` (*rs_A1* [*rs_A2*] *rs_B1* [*rs_B2*] [`distance` | `alignment`])
- returns the `real` contact area difference measure (described in `Abagyan and Totrov, 1997`) between two conformations A and B of the same set of residue pairs from two different objects. The set of residue pairs in each object (A or B) can be defined in two ways:

◊ by a single selection *rs_A1* : all pairs between selected residues (is equivalent to *rs_A1 rs_A1*)
◊ by two residue selections *rs_A1 rs_A2:* cross pairs between two sets of selected residues (e.g. the contacts between two subunits)

The measure is a normalized sum of differences between residue-residue contact areas in two conformations. The measure was calibrated on a set of pairs of conformations. The average distortion due to a noncrystallographic symmetry is about 5%, the average CAD between a pair of models in an NMR entry is 15%. Note that the paper uses an additional factor of 1.8 (i.e. CAD=1.8*Cad()) to bring the scale down to 0:100%, because about 40% of the contacts are trivial contacts between the neighboring residues. However, in evaluation of the docking solutions coefficient 1.8 should not be used. Loops are somewhat intermediate, but still a coefficient of 1.8 is recommended for consistency. The whole matrix of contact area differences is returned in `M_out` . This matrix can be nicely plotted with the `plot area M_out number ..` command (see example). The full matrix can also be used to calculate the residue profile of the differences.

See also: `Area`() function which calculates absolute residue-residue contact areas.
Options:
◊ `distance` option allows one to compare approximations of the inter-residue contact areas by the Ca and Cbeta positions. This allows one to calculated deformations between two homologous proteins which is not possible in the default mode in which two chemically identical molecules are compared. The residue pairs in two homologs are equivalenced according to the alignments `link`ed to the molecules. Residues deleted in a homologue are considered to have zero contact.
◊ `alignment` option is described in Marsden, Abagyan, 2004, Bioinformatics, v20, 2333-2344.

Examples:

```
# Ab initio structure prediction, Overall models by homology
   read pdb "cnf1"   # one conformation of a protein
   read pdb "cnf2"   # another conformation of the same protein
   show 1.8*Cad(a_1. a_2.)   # CAD=0. - identical; =100. different
   show 1.8*Cad(a_1.1 a_2.1) # CAD between the 1st molecules (domains)
   show 1.8*Cad(a_1.1/2:10 a_2.1/2:10) # CAD in a window
   PLOT.rainbowStyle = 2
   plot area grid M_out comment=String(Sequence(a_1,2.1)) link display

# Loop prediction: 0% - identical; ~100% totally different
#    CAD for loop 10:20 and its interactions with the environment
   show 1.8*Cad(a_1.1/10:20 a_1.1/* a_2.1/10:20 a_2.1/*)
#    CAD for loop 10:20 itself
   show 1.8*Cad(a_1.1/10:20 a_1.1/10:20 a_2.1/10:20 a_2.1/10:20)

# Evaluation of docking solutions: 0% - identical; 100% totally different
   read pdb "expr"   # one conformation of a complex
   read pdb "pred"   # another conformation of the same complex
   show Cad(a_1.1 a_1.2 a_2.1 a_2.2)  # CAD between two docking solutions
#
# ANOTHER EXAMPLE: the most changed contacts
   read object "crn"
   copy a_ "crn2"
   randomize v_ 5.
   Cad(a_1. a_2.)
   show s_out
   read column group input= s_out name="cont"
   sort cont.1
   show cont
# the table looks like this (the diffs can be both + and -):
#>T cont
#>-1-----------2-----------3----------
   -39.        a_crn.m/38  a_crn.m/1
   -36.4       a_crn.m/46  a_crn.m/4
   -32.1       a_crn.m/46  a_crn.m/5
   -29.8       a_crn.m/30  a_crn.m/9
   -25.2       a_crn.m/37  a_crn.m/1
...
   42.5        a_crn.m/43  a_crn.m/5
   45.1        a_crn.m/44  a_crn.m/6
   45.2        a_crn.m/43  a_crn.m/6
   55.3        a_crn.m/46  a_crn.m/7
   56.         a_crn.m/45  a_crn.m/7
```

Comparing two different, but structurally homologs proteins, via residue-residue contact conservation.

`Cad` (*rs_A1* [*rs_A2*] *rs_B1* [*rs_B2*] `alignment`)

Ceil

rounding function.
`Ceil` (*r_real* [*r_base*])
- returns the smallest `real` multiple of *r_base* exceeding *r_real.*
`Ceil` (*R_real* [*r_base*])
- returns the `rarray` of the smallest multiples of *r_base* exceeding components of the input array *R_real.* Default *r_base*= 1.0 .
See also: `Floor(`).

Cell

crystallographic cell function.
`Cell` ({ *os* | `m_map` })
- returns the `rarray` with 6 cell parameters {a,b,c,alpha,beta,gamma} which were assigned to the object or the map.

Charge

returns an rarray of partial electric charges of selected atoms, or total charges for residue, molecule or objects, depending on the selection level.
Charges can also be shown with a regular `show` *as_select* command.
`Charge` ({ *os* | *ms* | *rs* | *as* } [`formal` | `mmff`])
- returns `rarray` of elementary or total charges depending on the selection level.
  - ◊ `formal` : return formal charges
  - ◊ `mmff` : return *formal* charges calculated according to *mmff* atom types and rules. **Note:** do not confuse this option with a function to return the mmff charges.

Examples:

```
 build string "ala his glu lys arg asp"
 show Charge(a_1)      # charge per molecule
 show Charge( a_1/* )  # charge per residue
 show Charge( a_1//* ) # charge per atom

 avC=Charge(a_/5)         # total electric charge of 15th residue
 avC=Sum(Charge(a_/5/*))  # another way to calculate it
 show Charge(a_//o*)      # array of oxygen charges

# to return mmff charges:
 set type mmff
 set charge mmff
 Charge( a_//* )

# to return total charges per molecular object:
 read mol s_icmhome+"ex_mol.mol"
 set type mmff
 set charge mmff
 Charge( a_*. )
```

See also: `set charge`.

Chemical function. Converting and Generating library compounds.

Converting 3D objects to chemical arrays.

`Chemical(` *ms|os* `[exact]` `[hydrogen]` `[unique]` `[pharmacophore]`) returns an array of chemicals from a molecular selection of 3D molecular objects, e.g. `a_H` for hetero-molecules By default the selected molecules will be converted to 2D graphs. However with the `exact` option the original 3D coordinates will be retained in the elements of the chemical array. If you want to preserve explicitly drawn hydrogens `hydrogen` option should be used. Note that the number of chemicals in the array will be determined by the **selection level**. At the object level multiple molecules of the same object will be merged into one array element. With `unique` option duplicates will be excluded from the result.

Example:

```
read pdb "1ch8"
group table t_2D Chemical(a_H)   # convert to 2D chemical table
group table t_3D Chemical(a_H exact)  # make 3D chemical table without hydrogens
group table t_3D_hyd Chemical(a_H exact hydrogen) # make 3D chemical table, keep
```

With `pharmacophore` option the function generates pharmacophore points for the input selection.

Example:

```
read object s_icmhome + "biotin.ob" name="biotin"
read mol input = String(  Chemical(a_ pharmacophore )) name="biotin_ph4"
display xstick
display wire a_biotin.
```

To display supported pharmacophore types and use `show pharmacophore type` command

Converting smiles to chemical arrays:

`Chemical(` `S_smiles`|`s_smiles` `)`

returns an array of chemicals from a string arrays of smiles.

Example:

```
add column t Chemical({"N[C@@](F)(C)C(=O)O", "C[C@H]1CCCCO1"})
```

Converting InChI to chemical arrays:

`Chemical(` `S_InChI`|`s_InChI` `)`

See also: `chemical functions`

Generating combinatorial compounds from a Markush structure and R-group arrays.

`Chemical(` *scaffold I_RgroupNumArray* `enumerate` `)`   `returns` `one chemical`

The *I_RgroupNumArray* is an array of as many elements as there are different R groups in the *scaffold*.E.g. if there is R1 R2 R3 than this parameter can be {10,21,8}. The numbers refer to the R-group arrays linked to the *scaffold*.E.g.

```
group table scfld Chemical("C(=CC(=C(C1)[R2])[R1])C=1") "mol"
link group scfld.mol 1 Chemical({"N","O","S"})
link group scfld.mol 2 Chemical({"[Cl]", "[C*](=O)O"})

Nof( scfld.mol library ) # returns the total number of molecules in that combina
Nof( scfld.mol group )   # returns an array of sizes of each linked array in R1

Chemical( scfld.mol {1 1} enumerate )
Chemical( scfld.mol {1 2} enumerate )
Chemical( scfld.mol {2 2} enumerate )
```

`Chemical(` `enumerate` *scaffold* [`simple`] *R1 R2 ...* `)`   `returns` `enumeration result`

The same as above but does not require explicit linkage with `link group` command.

Example:

```
Chemical( enumerate  Chemical("C(=CC(=C(C1)[R2])[R1])C=1") Chemical({"N","O","S'
```

simple mode is similar to `enumerate library` and requires that size of R-group arrays be the same.

Example:

```
Chemical( enumerate  Chemical("C(=CC(=C(C1)[R2])[R1])C=1") simple Chemical({"N",
```

See also: `linking` scaffold to R-group arrays and the `Nof`

Cluster

`Cluster(` *I_NxM_NearestNeighb i_M_totalNofNearNeighbors i_minNofCommonNeighbors*)   *I_N_clusterNumbers*
function returns `iarray` of cluster numbers for each or N points.
The input to the first function is an array of M nearest neighbors (defined by the second argument *i_M_totalNofNearNeighbors)* for each of N points. For example for an array for 5 points, and i_M_totalNofNearNeighbors = 3 it can be an array like this: `{3,4,5, 1,3,4 1,2,5 2,3,5 1,2,3}` . The points will be grouped into the same cluster if the number of neighbors they share is larger or equal than *i_minNofCommonNeighbors* . This clustering algorithm is adaptive to the cluster density and does not depend on absolute distance threshold. In other words it will identify both very sparse clusters and very dense ones. The nearest neighbor array can be calculated by the with the Link(*I_bitkeys* , *nBits, nNearestNeighbors*) function.
`Cluster(` *M_NxNdist r_maxDist*)   *I_N_clusterNumbers*
This function identifies the *i_totalNofNeighbors* nearest neighbors from the full distance matrix *M_NxNdist* for each point and assembles points sharing the specified number of *common* neighbors in clusters.
All singlets (a single item not in any cluster) are placed in a special cluster number **0** . Other items are assigned to a cluster starting from 1.
Example with a distance matrix:

```
# let us make a distance matrix D
# we will cook it from 5 vectors {0. 0. 0.}
 m=Matrix(5,3)  # initialize 5 vectors
 m[2,1:3]={1. 0. 0.}   # v2
 m[3,1:3]={1. 1. 0.}   # v3
 m[4,1:3]={1. 1. 1.}   # v4
 m[5,1:3]={1. 0.1 0.1} # v5 close to v2

 D = Distance( m )  # 5x5 distance matrix created

 Cluster( D , 0.2 ) # v2 and v5 are assigned to cluster 1
 Cluster( D , 0.1 ) # radius too small. All items are singlets
 Cluster( D , 2.  ) # radius too large. All items are in cluster 1
```

Collection

The function to create a `collection` object

`Collection()` - returns empty `collection` object

`Collection(` *s_json_string*) - returns a `collection` object from a text in `JSON` format

`Collection(` *s_url_encoded_string*) - returns a `collection` object from a URL encoded string ("a=1&b=abc")

`Collection( web`) - returns a `collection` object from the POST or GET arguments. Can be used in CGI scripts. Multi-part content is also supported.

`Collection(` *table_row*) - returns a `collection` object for the table row.
Collection(t[1])

`Collection(` *table* `{column|header}`) - converts table columns or header part to the `collection`

`Collection(` *table|tab_column* `format`) - returns a `collection` object with the members controlling format, color and function for calculated columns. This collection can be modified and set back to the table or table column with the `set format` *collection* command . Example:

```
add column t {1 2 3} {1 2 3}
add column t function= "A+B"
set format t.A "<i>%1</i>"
show format t
c = Collection(t.A format )  # modify c
set format t.B c
```

Color

returns RGB values or color names.

`Color` (*as_n* `ball|cpk|label|skin|surface|wire`)   *M_nx3_rgb* - returns an rgb `matrix` of colors for a particular representation (0. 0. 0. 0. means black or undisplayed)
`Color` (*g_grob*)   *M_nx3_rgb* - returns `matrix` of RGB numbers for each vertex of the `g_grob` (dimensions: `Nof` (*g_grob)*,3).
See also: `color grob matrix`.
Example:

```
 build string "se his"
 display xstick
 make grob image name="g_"
 display g_ only smooth
 M_clr = Color( g_ )
 for i=1,20    # shineStyle = "color" makes it disappear completely
   color g_ (1.-i/20.)*M_clr
 endfor
 color g_ M_clr
```

`Color(` *M_rgb*)   *S_colorNames*

- returns sarray of color names approximating the rgb values in the matrix. The color names and definitions are taken from the `icm.clr` file. Example:

```
m = Matrix(3)
Color(m) # returns {"red","blue1","green"}
```

`Color( system )`

- returns `sarray` of system color names.

`Color( system i_numColor )`

- returns a name of a system color by number.

Example:

```
N = Nof( Color( system ))
for i=1,10
  print Color( system Random(N) )   # randomly pick one color
endfor
```

`Color( background )`
- returns `rarray` of three `RGB` components of the background color.

Interpolating colors by gradient

`Color(` *r_value s_gradient* [*r_from r_to*])   *R_3rgb* - returns 3-element rarray with `RGB` components describing the color and useful for the `color .. rgb=` command.

`Color(` *R_N_values s_gradient* [*r_from r_to*]) - returns matrix with *N* rows and 3 columns where each row is the `RGB` representation of the interpolated color for the respective value in the *R_N_values* array.

Examples:

```
s = "red/lime/blue"
Color( 0.  s 0. 1. )
Color( 0.5 s 0. 1. )
Color( 1.0 s 0. 1. )
Color( 0.1 s 0. 1. )
Color( 0.8 s 0. 1. )
Color( {0.1 0.8} s 0. 1. )
Color( {1. 8.} s 0. 10. )

Color( 0.1 "red/lime/blue,0:1" )
Color( {0.1 0.8} "red/lime/blue,0:1" )
Color( {1. 8.} "red/lime/blue,0:10" )
```

Image color functions

`Color(` *imageArray* `background`)

returns `sarray` with background colors of the images in *imageArray_*.The color of the top left pixel of the image is returned as the background color currently.

See also: `Image`, `image parray`

Consensus

`Consensus` (*ali*)   *s_consensus*
- returns the `string` consensus of alignment *ali_*. The `consensus characters` are these: # hydrophobic; + RK; - DE; ^ ASGS; % FYW; ~ polar. In the `selections by consensus` a letter code (h,o,n,s,p,a) is used.
`Consensus` (*ali* { *i_seq* | *seq* })
- returns the `string` consensus of alignment *ali_* as projected to the sequence. Sequence can be specified by its order number in the alignment or by name.
Example displaying conserved residues:

```
 read alignment "sx"  # load alignment
 read pdb "x"         # structure
 display ribbon
        # multiply rs_ by a mask like "  A C   N  .."
 cnrv = a_/A & Replace(Consensus(sx cd59),"[.^~#]"," ")
 display cnrv red
 display residue label cnrv
```

`Consensus` (*ms*|*rs*)

surface accessible areas projected on the selected residues via linked sequence and alignment.

Contour

making a table with the contour lines of a 2D function represented by a matrix for display in the `plot` command.

`Contour(` *M* [*r_step*|*i_numContours* [*fmin,fmax*]] [*R_Xs*|*R_Ys*])   `T_contourData` (X,Y,conn,Z)

Example (UNFINISHED):

```
M = 10.*Smooth(Smooth(Smooth(Matrix(100))))
tt = Contour(M,10,0.,5.)
delete tt.Z == 0.
sort tt.Z
add column tt "_black line 0.5" name="mark"
```

```
plot tt.X tt.Y tt.mark "/tmp/tmp.eps" append
```

Corr

linear correlation function (Pearson's coefficient **R**)
`Corr` (*R_X*, *R_Y*) `r_correlation`
- returns the `real` value of the linear correlation coefficient. Probability of the null hypothesis of zero correlation is stored in `r_out` .

Note: this function returns **R** , not $\mathbf{R^2}$. Taking it to the 2nd power can be a humbling experience.
Examples:

```
r=Corr(a,b)                 # two vectors a and b
if (Abs(r_out) < 0.3) print "it is actually as good as no correlation"
```

See also: `LinearFit`( ) function.

Cos

cosine function. Arguments are assumed to be in degrees.
`Cos` ({ *r_Angle* | *i_Angle* })
- returns the `real` value of cosine of its real or integer argument.
`Cos` (*rarray*)
- returns `rarray` of cosines of each component of the array.
Examples:

```
show Cos(60.)                            # returns 0.5
show Cos(60)                             # the same

rho={3.2 1.4 2.3}                        # structure factors
phi={60. 30. 180.}                       # phases
show rho phi rho*Cos(phi) rho*Sin(phi)   # show in columns rho, phi,
                                         # Re, Im
```

Cosh

hyperbolic cosine function.
`Cosh` ({ *r_Angle* | *i_Angle* }) - returns the `real` value of hyperbolic cosine of its real or integer argument. ***$Cos(x)=0.5(e^{iz} + e^{-iz})$***
`Cosh` (*rarray*) - returns `rarray` of hyperbolic cosines of each component of the array.
Examples:

```
show Cosh(1.)                            # 1.543081
show Cosh(1)                             # the same

show Cosh({-1., 0., 1.})                 # returns {1.543081, 1., 1.543081}
```

Count

function creates an iarray. Summary:

- ◊ `Count(` *i_n* `)`   *I_1,2,3,..n*
- ◊ `Count(` *i_from i_to* [*i_step=1*] `)`   *I_from,...,to*
- ◊ `Count(` *I*|*R*|*S_array* `)`   *I_1,..,n*
- ◊ `Count(` *I*|*R*|*S* `unique`|`identity`|`number)`   *I* # `111222233 or 123123412 or 333444422`

Detailed descriptions:
`Count` ([*i_Min*,] *i_Max*) - returns `iarray` of numbers growing from *i_Min* to *i_Max*. The default value of *i_Min* is 1.
Examples:

```
show Count(-2,1)          # returns {-2,-1,0,1}
show Count(4)             # returns {1,2,3,4}
```

See also the `Iarray( )`.
`Count` (*array*)
- returns `iarray` of numbers growing from 1 to the number of elements in the *array*.

`Count` (*I|R|S_array* `unique` | `identity`) `I`

returns an integer array with integer id for sequentially identical values.
Example:

```
group table t {"d","d","d","bb","bb","a","a","a"}
add column t Count(t.A unique ) Count(t.A identity ) name={ "unique","ide
show t
 #>T t
 #>-A-----------unique------identity---
    d           1           1
    d           1           2
    d           1           3
    bb          2           1
    bb          2           2
    a           3           1
    a           3           2
    a           3           3
```

CubicRoot

`CubicRoot(` *r* `)` *r_cubic_root*

`CubicRoot(` *r* [*r_im*] `)` *R6_3re+3im*

Example:

```
CubicRoot(27. )
  3.
CubicRoot(27. 0.)
 #>R
   3.
   -1.5
   -1.5
   0.
   -2.598076
   2.598076
```

See also: SolveCubic, `Sqrt`

Date

Summary:

`Date( )` `e_1currentDate`

`Date(` *n* `)` `e_arrayOf_n_currentDates`

returns an `date array` of current system date and time.

Example:

```
print "Today is :" Date()
```

`Date ( version )` `e_dateOfCompilation`

`Date (` *os* `)` `e_pdbDates`

returns the date of the pdb file creation in an `date array` format. The date read from the HEADER record of a pdb file and is stored with the object.

Example:

```
read pdb "1crn"
if Date(a_) > Date("1980","%Y") print "released after 1980"
```

`Date` ({*s_date*|*S_dates*} [*s_format*])

converts `string` or `sarray` to dates using *s_format* or default TOOLS.dateFormat

Example:

```
String( Date( "12 Oct 2002", "%d %b %Y" ) "%Y-%m-%d" )
```

The allowed format specifications are the following:

format	description
%a or %A	The weekday name according to the current locale, in abbreviated form or the full name.
%b or %B or %h	The month name according to the current locale, in abbreviated form or the full name.
%c	The date and time representation for the current locale.
%C	The century number (0-99).
%d or %e	The day of month (1-31).
%D	Equivalent to %m/%d/%y. (This is the American style date, very confusing to non-Americans, especially since %d/%m/%y is widely used in Europe.)
%H	The hour (0-23).
%I	The hour on a 12-hour clock (1-12).
%j	The day number in the year (1-366).
%m	The month number (1-12).
%M	The minute (0-59).
%n	Arbitrary whitespace.
%p	The localeâ s equivalent of AM or PM. (Note: there may be none.)
%r	The 12-hour clock time (using the localeâ s AM or PM). (%I:%M:%S %p)
%R	Equivalent to %H:%M.
%S	The second (0-60; 60 may occur for leap seconds; earlier also 61 was allowed).
%t	Arbitrary whitespace.
%T	Equivalent to %H:%M:%S.
%U	The week number with Sunday the first day of the week (0-53). The first Sunday of January is the first day of week 1.
%w	The weekday number (0-6) with Sunday = 0.
%W	The week number with Monday the first day of the week (0-53). The first Monday of January is the first day of week 1.
%x	The date, using the localeâ s date format.
%X	The time, using the localeâ s time format.
%y	The year within century (0-99). When a century is not otherwise specified, values in the range 69-99 refer to years in the twentieth century (1969-1999); values in the range 00-68 refer to years in the twenty-first century (2000-2068).
%Y	The year, including century (for example, 1991).

Example:

```
String( Date() "%b %d %Y %I:%M%p" )  # Current date and time in American
String( Date() "%d/%b/%Y %H:%M" )    # European style
```

Deletion

`Deletion` (*rs_Fragment*, *ali_Alignment* [, *seq_fromAli*] [, *i_addFlanks*] [{`"all"`|`"nter"`|`"cter"`|`"loop"`}])
- returns the `residue selection` which flanks deletion points from the

viewpoint of other sequences in the *ali_Alignment.* If argument *seq_fromAli* is given (it must be the name of a sequence from the alignment), all the other sequences in the alignment will be ignored and only the pairwise sub-alignment of *rs_Fragment* and *seq_fromAli* will be considered. The alignment must be `linked` to the object. With this function (see also `Insertion` function) one can easily and quickly visualize and/or extract all *indels* in the three-dimensional structure. The default *i_addFlanks* parameter is 1. String options:

· "all" (default: no string option) select deletions of all types
· "nter" select only N-terminal fragments
· "cter" select only C-terminal fragments
· "loop" select only the internal zones of deleted loops

See example coming with the `Insertion( )` function description.

Descriptor

`Descriptor` (*chemArray predModel*)

- returns vector of `rarrays` with chemical descriptors calculated for each chemical. each `rarray` consists of chemical fingerprint part and values for columns with formula used in the *predModel.*

This information can be used for further analysis or exported outside ICM.

Example:

```
 # assumes that 'clogPpred' is a prediction model
 tt = Table( Transpose( Matrix( Descriptor( Chemical("CCC"), clogPpred )
 add column tt Name( clogPpred column )
 sort reverse tt.A
```

To find the description of the each particular position in the rarray Name function can be used.

Example:

```
rr = Descriptor( Chemical("CCC") myModel )[1]
na = Name( myModel column )
for i=1,Nof(rr)
  if (rr[i] != 0) print rr[i], na[i]
endfor
```

See also: Name(predModel, column .

Det

determinant function.
`Det` (*matrix*)
- returns a `real` determinant of specified square *matrix.*
Examples:

```
 a=Rot({0. 0. 1.}, 30.)      # Z-rotation matrix by 30 degrees
 print Det(a)                # naturally, it is equal to 1.
```

Disgeo

Solves the so called `"DIStance GEOmetry"` problem (finding coordinates from a distance set). This function can be used to visualize in two or three dimensions a distribution of homologous sequences:

```
 group sequence se1 se2 se2 se4 mySeqs
 align mySeqs
 distMatr=Distances(mySeqs)
```

or any objects between which one can somehow define pairwise distances. Since

principal coordinates are sorted according to their contribution to the distances and we can hardly visualize distributions in more than three dimensions, the first two or three coordinates give the best representation of how the points are spread in n-1 dimensions. Another application is restoring atomic coordinates from pairwise distances taken from NMR experiments.
`Disgeo` (*matrix*)
- returns `matrix` *[1:n*,1:n] where the each row consists of n-1 coordinates of point *[i]* sorted according to the eigenvalue (hence, their importance). The first two columns, therefore, contain the two most significant coordinates (say X and Y) for each of *n* points. The last number in each row is the eigenvalue *[i]*. If distances are Euclidean, all the eigenvalues are positive or equal to zero. The eigenvalue represents the "principal coordinate" or "dimension" and the actual value is a fraction of data variation due to the this particular dimension. Negative eigenvalues represent "non-Euclidean error" in the initial distances.

- `R_out` returns *four* numbers: total negative eigen values, and the first 3 largest positive eigenvalues. All scaled to 100%.

Example:

```
read sequences s_icmhome+"zincFing"   # read sequences from the file,
list sequences               # see them, then ...
group sequence alZnFing      # group them, then ...
align alZnFing               # align them, then ...
a=Distance(alZnFing)         # a matrix of pairwise distances
n=Nof(a)                     # number of points
b=Disgeo(a)                  # calculate principal components
corMat=b[1:n,1:n-1]          # coordinate matrix [n,n-1] of n points
eigenV=b[1:n,n]              # vector with n sorted eigenvalues
xplot= corMat[1:n,1]
yplot= corMat[1:n,2]
plot xplot yplot CIRCLE display # call plot a 2D distribution
```

Distance

generic distance function. Calculates distances between two ICM-shell objects, bit-strings or molecular objects, or extracts distances from complex ICM-shell objects.

`Distance(` *II* | *RR* | *as as* | *seq seq*)   *r_dist*

`Distance(` *S*|*s*, *s*)   *R*|*r*

`Distance(` *ali ali* `[exact]`)   *r*

`Distance(` *S S* `[simple]`)   `M`

`Distance(` *Mnk*)   *Mnn*

`Distance(` *Mnk Mmk*)   *Mnm*

`Distance(` *M_xyz*|*as M_xyz*|*as r_dist*)   *l_yes_if_closer_than_dist*

`Distance(` *seq seq*
`[identity|evolution|new|fast|number|reverse]`)

`Distance(` *seq seq* `nucleotide [len]`)

`Distance(` *seqArr[n]>)*   *<M_nn*

`Distance(` *ali seq* `[string]`)   `R_n_seq_in_ali`

`Distance(` *seqArr[n]> <seq*)   *R_n*

`Distance(` *seqArr[n]> <seqArr[m]>)*   *<M_nm*

`Distance(` *bitvecArr[n]> <bitvecArr[m]>)*   *<M_nm* `#tanimoto`

`Distance(` *as* [*r_default*=`-1.`] `)` *R_tether_lengths_or_def*

`Distance(` *as_n as_m* `)` `d_between_centers_of_mass`

`Distance(` *as_n as_m* `all )` *R_nm*

`Distance(` *as_n as_n* `rarray )` *R_n* `# aligned arrays, same n`

`Distance(` *ali* `[0] )` `M_interSeqDist`

`Distance(` *X_n* [*X_m*] `[pharmacophore] )`
`M_nxm_chemical_Tanimoto_distances`

`Distance(` *I_keys1 I_keys2 i_nBits|R_nbitWeights* `[simple] )` `M: Tanimoto|weighted`

`Distance(` *tree* [*i_at*=1] `split )` *r_splitLevel*

`Distance(` *tree* `all|modify )` *R_splitLevels|splitLevelTStats*

`Distance(` *g* `wire|grid` [*i_maxDist(1000000)>])* *<M_shortestPaths*

See detailed descriptions below.

Distance between iarrays

`Distance ( iarray1, iarray2 )`
- returns the `real` **sqrt** of sum of ***$(I1_i - I2_i)^2$*** .

Distance between vectors

`Distance (` *R_X*, *R_Y* `)` - returns the `real` Cartesian distance between two vectors of the same length. ***$D = Sum((X_i - Y_i)^2)$***

Distance ~~as_

`Distance (` *as_1*, *as_2* `[ all ] )`

- returns the `real` distance in Angstroms between centers of mass of the two specified selections. The interactive usage of this function: Option `all` will return an array of all cross distances between the selections. The selected virtual atoms will be skipped if the selection level residue, molecule or object. Othewise, if you explicitly select virtual atoms, they will be included, e.g.

```
build string "ala" # contains 2 virtual atoms at N terminus
build string "his" # also contains 2 virtual atoms at N terminus
Distance( a_1. a_2. all ) # no virtual atom distances
Distance( a_1.// a_2.// all ) # selected virtual atoms are included

Distance( a_1. a_2. ) # a single distance between centers of mass
```

Distance ~~as_ rarray

`Distance (` *as_1* , *as_2*, `rarray )`

- returns the `rarray` of distances in Angstroms between the two specified selections containing the same number of atoms (1-1, 2-2, 3-3, ...).

See also: `Distance (` *as1 as2* all `)`

Distance matrix

`Distance (` *M_coor* `)` - returns the square `matrix` of distances between the rows of the input matrix *M_coor.* Each row contains *m* coordinates (3 in 3D space). For example: `Distance(Xyz(a_//ca))` returns a square matrix of Ca-Ca distances.

Tanimoto distance between two arrays of bit-strings

`Distance(` *X_chem_n X_chem_m* `)` *M_nxm_distances*

`Distance(` *I_keys1 I_keys2* `nBits` | *R_nBitWeights* [`simple`])
M_distances
- returns the `matrix` of Tanimoto distances between two arrays of bit-strings. Each array of N-strings is represented by an `iarray` *I_keys* of *N*(nBits/32)* elements (e.g. if *nBits* is `32` , each integer represents 1 bit-string, if *nBits* i 64, I_keys1 has two integers for each bit string, etc.). The returned matrix dimensions are *N1 x N2* . The **distance** is defined as *1.* - **similarity** , where The **Tanimoto similarity** between bitstrings is defined as follows: The number of the *on*-bits in-common between two strings divided by the number of the *on*-bits in either bit-string.
You can provide a relative weight for each bit in a bit-string as a `rarray` R_weights. In this case the **weighted Tanimoto distance** is calculated as follows:

```
  distWeighted = 1. - Sum( Wi_of_common_On_Bits ) / Sum( Wi_of_On_Bits )
```

With option `simple` the similarity calculation is modified so that the number of bits in common is divided by the number of bits in the **second** bit-string. For example:

```
Distance({3} {1} 32 simple ) # returns 0.
Distance({1} {3} 32 simple ) # returns 0.5
```

Example:

```
Distance({1 2 3},{1 2 3},32)
 #>M
 0. 1. 0.5
 1. 0. 0.5
 0.5 0.5 0.
```

The diagonal distances are 0; no bits are share between 1 (100..) and 2 (010..) (distance=1.) and one of two bits is shared between 1 (100..) and 3 (110..). Instead of the number of bits, one can provide the relative weights for each bit. The dimension of the bit-weight array then becomes the size of the bit-string. The weighted Tanimoto is calculated.

See also:

- Iarray-bits-to-integers{ Iarray({1 0 0 1 1 0 ..} key) } to generate compressed integer bit vectors

Distance matrix between two sets of coordinates

`Distance` (*M_coor1 M_coor2*) - returns the `matrix` of distances between the rows of the two input matrices. Each matrix row may contain any number of coordinates coordinates (3 in 3D space).
For example: **Distance(Xyz(a_/1:5/ca) Xyz(a_/10:12/ca)** returns a 5 by 3 matrix of distances between Ca-s of the two fragments.

`Distance(` *M_xyz1|as1 M_xyz2|as2 r_dist*) *l_yes_if_closer_than_dist* This function returns a logical `yes` if any two points or atoms in two sets of coordinates or selections are closer than the threshold.
`if Distance` (*as1 as2 r_dist*) `then` ...

is a more efficient version of this condition:

`if Nof( Sphere(` *as1 as2 r_dist*)) > 0

Distance tether

`Distance` (*as* [*r_defaultLength*=-1.])
- returns the `real array` of lengths of tethers for each selected atom or the default value (-1.). The default value can be set to any value. Tethers are assumed to be already set, see command `set tether`. Also note, that the expression `Distance( as_out )` will give the same results if `as_out` selection was not changed by another operation; see also `special selections`.
Example:

```
read pdb "1crn"
convert tether # keeps tethers to the pdb original
deviations = Distance( a_//!h*,vt* , 9.9)
perResDevs = Group( deviations, a_//!h*,vt* ,"max") # find max.devs per
display ribbon
color ribbon a_/* perResDevs

# Another example
 Distance( a_//T ) # selects only tethered atoms
 #>R
    1.677
    1.493
    1.386
    1.435
    1.645
    1.570
    2.165
    1.399
```

Distance Dayhoff

`Distance(` *seq1 seq2* `[identity|evolution|new|fast|number|reverse] )   r`

`Distance(` *seqArr*`[n]` *seq* `)`   *R_n*

- returns the `real` measure of similarity between two aligned sequences. Zero distance means 100% identity. The distance is calculated by the following two steps:

1. d1 = 1.0 - (nResidueIdentities/Min(Length(Seq1), Length(Seq2)) (d1 belongs to [0.,1.] range)
2. if there is no `identity` option the distance is corrected: Distance(Seq1,Seq2) = DayhoffTransformation(d1)

Transformation practically does not change small distances d1, whereas large distances, especially above 0.9 (10% sequence identity) are increased to take occasional reversals into account. Distances d1 within [0.9,1.0] are transformed to [5.17, 10.] range.

See also: `Distance (` *ali* `)` for distance and seq.identity matrices.

Distance between sequences or alignment sequences

`Distance` (*alignment*)   *M_nxn*

`Distance(` *seqArr_n*)   *M_nxn*

`Distance(` *seqArr_n seqArr_m*)   *M_nxm*

- returns `matrix` of pairwise sequence-sequence distances in the alignment. These distances are calculated with the `fast` option as follows

```
 1.-(nResidueIdentities-gapPenalty)/Min(Length(Seq1), Length(Seq2))
```

where gapPenalty is 3 for each gap.
Without the `fast` option the distances are calculated based on comparison matrix and gap penalties. These distances are more sensitive but there is no

simple mapping between them and percent identity based distances.
Example:

```
 read alignment msf s_icmhome+"azurins"            # read azurins.msf
 NormCoord = Disgeo(Distance(azurins))   # 2D sequence diversity in
#
# calculate pairwise sequence identities
 read alignment "aln" name="aln"
 n=Nof(aln)
 mids = 100*(Matrix(n,n,1.) - Distance(aln ))  # the pairwise seq. identi
 t = Table( mids, Name(aln), Name(aln) )  # to convert the matrix into pa
 t = Table( mids, index )  # a simpler version with i,j
```

Distance between two alignments

`Distance` (*ali_1 ali_2* [`exact`])

- returns the `real` distance between two alignments formed by the same sequences.
The distance is defined as a number of non-gap columns identical between two alignments.
Two different normalizations are available:
The default normalization is to the shorter alignment. (`Distance` (*ali_1 ali_2*)). In this case the number of equivalent pairs is calculated and is divided by the total number of aligned pairs in the shorter alignment. *This method detects alignment shifts* but does not penalize un-alignment of previously aligned residue pairs. **D = (La_min - N_commonPairs)/La_min** In the following alignment the residue pairs which are aligned in *both alignments* are the same, therefore the distance is 0.

```
 show a1   # La1 = 3
 ABC---XYZ
 ABCDEF---
 show a2   # La2 = 6
 ABCXYZ
 ABCDEF
 Distance(a1,a2)   # a1 is a sub-alignment of a2, distance is 0.
 0.
```

exact option: normalization to the number of pairs of the longer alignment. By *longer* we mean the larger number of aligned pairs regardless of alignment length (the latter includes gaps and ends). **D = (La_max - N_commonPairs)/La_max** Now in the above example, La_max = 6 , while N_commonPairs = 3, the distance is 0.5 (e.g. the alignments are 50% different).

```
 Distance(a1,a2,exact)  # returns 0.5 for the above a1 and a2
```

Example showing the influence of gap parameters:

```
 read sequence msf s_icmhome+"azurins.msf"
 gapOpen =2.2
 a=Align(Azu2_Metj  Azup_Alcfa)    # the first alignment
 gapOpen =1.9                      # smaller gap penalty and ..
 b=Align(Azu2_Metj  Azup_Alcfa)    # the alignment changes
 show 100*Distance(a b )           # 20% difference
 show 100*Distance(a b exact )     # 21.7% difference
 show a b
```

The distance of the cluster splitting level

`Distance(` *treeArr i_at* `separator`)

- return the current value of the cluster splitting level set by `split` command.

Chemical similarity distance

`Distance(` *chemarray* `[pharmacophore] )`

- return square `matrix` of chemical distances. The chemical distance is defined as the Tanimoto distance between binary fingerprints Option `pharmacophore` uses different fingerprints based on ph4-type triplets.

Example:

```
Distance( Chemical( { "CCC", "CCO"} ) )
```

`Distance(` *chemarray1 chemarray2* `[pharmacophore] )`

- return a MxN `matrix` where *M* is `number of elements` in *chemarray1* and *N* is `number of elements` in *chemarray2* Option `pharmacophore` uses different fingerprints based on ph4-type triplets.

Example:

```
Distance( Chemical({ "CCC", "CCO"}) Chemical("CC" ))
```

Zero distance for non-identical compounds. Sometimes non-identical compounds can give a zero fingerprint distance due to the limitations inherent in finite length fingerprints. To make the distance more representative, one can mix different types of distances, e.g. for two chemical arrays `X1` and `X2`

```
Mdist = Distance( X1, X2 ) + 0.1*Distance(X1,X2, pharmacophore)
```

See also: `find table` `find molcart` `other chemical functions`

Eigen

eigenvalues/eigenvectors function.
`Eigen (` *M* `)`
- returns the square `matrix` (*n* x *n*) of eigenvector columns of the input *symmetric square* matrix *M_* . All *n* eigenvalues sorted by their values are stored in the `R_out rarray.`
Example:

```
A = Matrix(3, 3, 0.)       # create a zero square matrix...
A[1:3,1] = {1.,-2.,-1.}    # and set its elements
A[2,2] =  4.
for i = 1, 3-1             # the matrix must be symmetric
  for j = i+1, 3
    A[i,j] = A[j,i]
  endfor
endfor
X = Eigen(A)               # calculate eigenvectors...
V = R_out                  # and save eigenvalues in rarray V
printf "eigenvalue 1 eigenvalue 2 eigenvalue 3\n"
printf "%12.3f %12.3f %12.3f\n", V[1], V[2], V[3]
printf "eigenvector1 eigenvector2 eigenvector3\n"
for i = 1, 3
  printf "%12.3f %12.3f %12.3f\n", X[i,1], X[i,2], X[i,3]
endfor
```

Energy

function.
`Energy ( string )`
- returns the `real` sum of **pre-calculated** energy and penalty (i.e. geometrical restraints) `terms` specified by the string.
Important: this function does NOT calculate the energy, the terms must be **calculated** beforehand by invoking one of the following commands where energy is calculated at least once: `show energy`, `minimize`, `ssearch` command and `montecarlo` command.

Note:

- · Allowed terms in the string are "vw,14,hb,el,to,af,bb,bs,cn,tz,rs,xr,sf";
- · "func" stands for the total of all the terms, both energy and penalty;
- · "ener" is only the energy part (i.e. "vw,14,hb,3l,to,af,bb,bs,sf");
- · "pnlt" is only the penalty part (i.e. "cn,tz,rs,xr").
- · load conf and load frame commands fill out all the energy/penalty terms, which are stored in both stacks and trajectory files (of course the values also depend on a set of free variables). You can get the energy/penalty terms of the loaded conformation without explicitly recalculating them.

Examples:

```
read object s_icmhome+"dcLoop.ob"
show energy
print Energy("vw,14,hb,el,to")  # ECEPP energy

read stack s_icmhome+"dcLoop.cnf"
load conf 0
print Energy("func")  # extract the best energy without recalculating it
```

Energy (*rs* [simple | base | *s_energyTerms*])
- in contrast to the previous function this function with an explicit residue selection calculates and returns **residue** energies in an ICM object. convert the object if is not of the ICM type. The energies are calculated according to the current energy terms , and also depend on the fixation of the object. Use unfix only V_//S to restore standard fixation.
This function can be used to evaluate normalized residue energies for standard amino-acids to detect local problems in a model.

For normalized energies, use the simple option. The base option just shifts the energy value to the mean energy for this residue type. If the simple or base terms are *not* used, the current energy terms are preserved. The energies calculated with the simple or base option are calculated with the "vw,14,hb,el,to,en,sf" terms. The terms are temporarily enforced as well as the vwMethod = 2 and vwSoftMaxEnergy values, so that the normalization performed with the simple option is always correct.
This function will calculate residue energies for all terms and set-ups with the following exceptions:

- · electrostatic ("el") term and electroMethod = "boundary element", "MIMEL", or "generalized Born"

The *s_energyTerms* argument allows one to refine the energy terms dynamically (see example below).
Example:

```
read pdb "1crn"
delete a_W
convert
set terms "vw,14,hb,el,to,en,sf"
group table t Energy( a_/A ) "energy" Label(a_/A ) "res"
show t
unfix V_//*
group table tBondsAngles Energy( a_/A "bs,bb" ) "covalent" Label(a_/A )
show tBondsAngles
```

See also: the calcEnergyStrain macro.
Energy (conf *i_confNumber*)
- returns the table of all the energy components for a given stack conformations.
The table has two arrays:

- · sarray of the energy term names (.hd) and
- · rarray of energy values for each energy term (.ey) and

Energy ({ stack | conf })
- returns the rarray of total energies of stack conformations. Useful for

comparison of spectra from different simulations.
Examples:

```
read object s_icmhome+"crn.ob"
set terms only "vw,14,hb,el,to"  # set energy terms
show energy v_//xi*              # calculate energy with only
                                 # side chain torsions unfixed
         # energy depends on what variables are fixed since
         # interactions inside rigid bodies are not calculated,
         # and rigid body structure depends on variables

a = Energy("vw,14")      # a is equal to the sum of two terms

electroMethod="MIMEL"    # MIMEL electrostatics
set terms only "el,sf"   # set energy terms
show energy
print Energy("ener")     # total energy
print Energy("sf")       # only the surface part of the solvation energy
print Energy("el")       # electrostatic energy
print r_out              # electrostatic part of the solvation energy
```

Error

function indicates that the previous ICM-shell command has completed with error.
`Error`
- returns `logical yes` if there was an error in a previous command (not necessarily in the last one). After this call the internal error flag is reinstalled to `no.` The shell error flag can be set to `yes` with the `set error` command.

`Error( string )`
- returns `string` with the last error message. It also returns integer code of the *last* error in your script in `i_out` . In contrast to the logical `Error()` function, here the internal error code is **not** reinstalled to 0, so that you can use it in expressions like `if( Error ) print Error(string)` .
`Error(` *i_error_or_warning_code* `)` `l Error( number )` `s` - returns `logical yes` if an error or warning with the specified code occurred previously in the script. This call also resets the flag (e.g. `Error(415)`). This is convenient to track down certain warnings or errors in scripts (e.g. detecting if 'readpdb{read pdb} found certain problems).
Option `number` will return a `string` will previously set error and warning messages.
To clear all bits use the `clear error` command.

Examples:

```
read pdb "1mng"  # this file contains strange 28-th residue
if (Error) print "These alternative positions will kill me"

read pdb "1abcd"  # file does not exist
read pdb "1mok"
clear error
```

See also: `errorAction` , `s_skipMessages` , `l_warn`, `Warning`
`Error(` *r_x* `[ reverse ] )`
- returns `real` complementary error function of real ***x*** : ***erfc(x)=1.-erf(x))*** , defined as
(2/sqrt(pi)) integral{x to infinity} of exp(-t^2) dt
or its inverse function if the option `reverse` is specified. It gives the probability of a normally distributed (with mean 0. and standard deviation 1./Sqrt(2.)) value to be larger than *r_x* or smaller than *-r_x.*
Examples:

```
show 1.-Error(Sqrt(0.5)) # P of being inside +-sigma (about 68%)
show Error(2.*Sqrt(0.5)) # P of being outside +- 2 sigma
```

`Error(` *R_x* `)`
- returns `rarray` of ***erfc(x)=1.-erf(x))*** functions for each element of the real array (see above).

Examples:

```
x=Rarray(1000 0. 5. )
plot display x Error(x ) {0. 5. 1. 1. 0. 1. 0.1 0.2 }
plot display x Log(Error(x ),10.) {0. 5. 1. 1.}
      #NB: can be approximated by a parabola
      #to deduce the appr. inverse function.
      #Used for the Seq.ID probabilities.
```

Error (for SOAP messages)

`Error(` *soapMessage* `)`

- returns a error string from the SOAP message. (empty string if no error)

This function is used the check the result of calling SOAP method.

See: `SOAP services` for more details and examples.

Exist

function indicates if an ICM-entity exists or not.
`Exist` (*s_fileName* [`write` | `read` | `directory`]) - returns `logical yes` if the specified file or directory exists, `no` otherwise. Options:

- · `write` open for writing
- · `read` open for reading
- · `directory` the provided string is a directory (not file)

`Exist` (`key`, *s_keyName*) - returns `logical yes` if the specified keystroke has been previously defined. Examples: `Exist(key, "F1"`, `Exist( key, "Ctrl-B" )` See also: `set key` command.
`Exist` (`object`) - returns `logical yes` if there is at least one molecular object in the shell, `no` otherwise.
`Exist` (*os1* `stack`) - returns `logical yes` if there is a built-in object stack , `no` otherwise.
`Exist` (`box`) - returns `logical yes` if the purple box is displayed, `no` otherwise.
`Exist` (`view`) - returns `logical yes` if the GL - graphics window is activated, `no` otherwise.
`Exist` (`gui`) - returns `logical yes` if the GRAPHICS USER INTERFACE menus is activated, `no` otherwise.

`Exist` (*grob* `display`) - returns `logical yes` if the grob is displayed.
`Exist(` `connect`) - returns `logical yes` if the mouse rotations are `connected` to a graphical object (`grob`) or a molecular object.

`Exist(` *s_table_name* `sql table`) - returns `logical yes` if there is an `sql` table with the specified name exists. It works with the Molcart tables or tables accessed via the `Sql` function.

`Exist(` `variable` *s_varName*) - returns `yes` if the variable exists in the ICM shell, `no` otherwise. See also `Type(`). E.g.

```
Exist(variable, "aaa")  # returns no
aaa=234
Exist(variable, "aaa")  # returns yes
```

Examples:

```
if (!Exist("/data/pdb/") then
  unix mkdir /data/pdb
endif

if(!Exist(key,"Ctrl-B")) set key "Ctrl-B" "l_easyRotate=!l_easyRotate"

if !Exist(gui)  gui simple
```

`Exist(` *chemarray* `pattern`)

returns `logical` *yes* if at least one of the elements contains `SMARTS` search attributes, *no* - otherwise.

Example:

```
Exist( Chemical("[C&H1,N]") pattern )  # returns yes
Exist( Chemical("CCO") pattern ) # return no
```

Database information

`Exist(` *s_dbtable* `sql table`)

- returns `logical yes` if the specified table exists in the database

See also: `molcart`

Existenv

function indicating if an UNIX-shell environmental variable exists.
`Existenv` (*s_environmentName*)
- returns `logical yes` if the specified named environment variable exists.
Example:

```
 if(Existenv("ICMPDB")) s_pdb=Getenv("ICMPDB")
```

See also: `Getenv(`), `Putenv(`) .

Extension

function.
`Extension` (*string* [`dot`])
- returns `string` which would be the extension if the string is a file name. Option `dot` indicates that the dot is excluded from the extension.
`Extension` (*sarray* [`dot`])
- returns `sarray` of extensions. Option `dot` indicates that the dot is excluded from the extensions.
Examples:

```
 print Extension("aaa.bbb.dd.eee")   # returns ".eee"
 show Extension({"aa.bb","122.22"} dot)        # returns {"bb","22"}
 read sarray "filelist"
 if (Extension(filelist[4])==".pdb") read pdb filelist[4]
```

Exp

exponential mathematical function (e^x).
`Exp` (*real*)
- returns the `real` exponent.
`Exp` (*rarray*)
- returns `rarray` of exponents of *rarray* components.
`Exp` (`matrix`)
- returns `matrix` of exponents of *matrix* elements.
Examples:

```
 print Exp(deltaE/(Boltzmann*temperature))   # probability
 print Exp({1. 2.})                          # returns { E, E squared }
```

Field

function.
`Field` (*s* [*s_precedingString*] *i_fieldNumber* [*s_fieldDelimiter*])
- returns the specified field. Parameter `s_fieldDelimiter` defines the

separating characters (space and tabs by default). If the field number is less than zero or more than the actual number of fields in this string, the function returns an empty string.
The s_fieldDelimiter string
Single character delimiter can be specified directly, e.g.

```
 Field("a b c",3," ")  # space
 Field("a:b:c",3,":")  # colon
```

Alternative characters can be specified sequentially, e.g.

```
 Field("a%b:c",3,"%:")  # percent OR colon
```

Multiple occurrence of a delimiting character can be specified by *repeating* the same character *two* times, e.g.

```
 Field("a  b   c",3,"  ")     # two==multiple spaces in field delim
 Field("a%b:::c",3,"%::")    # a single percent or multiple colons
```

You can combine a single-character delimiters and multiple delimiters in one *s_fieldDelimiter* string.
More examples:

```
 s=Field("1 ener glu 1.5.",3)     # returns "glu"
 show Field("aaa:bbb",2,":")      # returns "bbb"
 show Field("aaa 12\nbbb 13","bbb",1) # returns "13"
 show Field("aaa 12\nbbb 13 14","bbb",2,"  \n\n") # two spaces and two \
# another example
 read object s_icmhome+"all"
       # energies from the object comments, the 1st field after 'vacuum'
 show Rarray(Field(Namex(a_*.),"vacuum",1))
```

`Field` (*S* , [*s_precedingString*] *i_fieldNumber* [*s_fieldDelimiter*])
- returns an `string array` of fields selected from *S_* string array .
`s_fieldDelimiter` is the delimiter. If the field number is less than zero or more than the actual number of fields in this string, an element of the array will be an empty string.
Examples:

```
 show Field({"a:b","d:e"},2,":") # returns {"b","e"}
 s=Field({"aa 2 3.3", "bb 4 1.3", "cc 31a 1.1 3"},2)
         # returns {"2","4","31a"}
 s=Field({"aa 2 3.3", "bb 4 1.3", "cc 31a 1.1 3"},4)
         # returns {"","","3"}
```

See also: `Split`( ).

User field from a selection

`Field`(*as*|*rs*|*ms*|*os* [*s_fieldName*])
`Field`({ *rs* | *ms* | *os* } [*i_fieldNumber*])

`Field`(*os* `15`)

returns `rarray` of user-defined field values of a selection. Some fields are filled upon `reading` a pdb file
Atoms. Only one user defined field can be set to atoms, e.g.

```
read object s_icmhome+"crn.ob"
set field a_//* Random(0.,1.,Nof(a_//*))
show Field( a_//* )

read pdb "1f88" # rhodopsin, many loops missing
Field( a_ 15) # returns 31. residues
Field( a_ "pmid") # iarray[1] with pubmed id, automatically created by re
set field a_/10,14,21 name="pocket"
display cpk Field ( a_/* "pocket" )
```

Residues, molecules and objects.
Three user fields can be defined for each residue and up to 16 for molecules and objects. To extract them specify *i_fieldNumber* . The level of the selection determines if the values are extracted from residues, molecules or objects. Use the selection level functions `Res` `Mol` and `Obj` to reset the level if needed. For example: `Res(Sphere(gg, a_1. 3.))` selects residues of the 1st object which are closer than 3. A to `grob` `gg` .

Upon reading a pdb file the object **field 15** contains the number of residues **missing** from the ATOM records, but present in `SEQRES` records due to local disorder. Example:

```
read object s_icmhome+"crn.ob"
set field a_/A Random(0.,1.,Nof(a_/A)) number = 2 # set the 2nd field to
color a_/* Field( a_/A  2 )                       # color by it
```

Standard fields:

· **object:** "pmid" - integer pubmed id

See also:
· `set field` *as_* [name= *s*] .. ,
· `Smooth` *rs_* to generate 3D-averaged user fields
· `Select` function to select by user defined field (e.g. Select(a_// "x>-1.")).

File

function returning file names or attributes of named files.
`File` (*os*) returns the name of the source file for this object. If the object was created in ICM or did not come from an object or PDB file, it returns an empty string.
Example:

```
read pdb "/home/nerd/secret/hiv.ob"
File( a_ )
 /home/nerd/secret/hiv.ob
```

`File` (*s_file_or_dir_Name* `"length"`)
- returns `integer` file size or -1.
`File` (*s_file_or_dir_Name* `"time"`)
- returns `integer` modification time or -1. Useful if you want to compare which of two files is newer.

`File` (`icm_object`)
- returns `string` file name from which this object has been loaded or empty string.
`File` (*s_file_or_dir_Name*)
- returns `string` with the file or directory attributes separated by space.

Note that this function will only work on Unix or Mac, see a`Exist (*s_file* ..) function for cross-platform functions. If file or directory do not exist the function returns **"- - - - 0"** Otherwise, it contains the following 4 characters separated by space and the file size:

1. type character:
   - 'f' - regular file
   - 'd' - directory
   - 'l' - symbolic link
   - 'c' - character special file
   - 'p' - pipe
2. 'r' if you can read the file (or from the directory)
3. 'w' if you can write to this file (or directory)
4. 'x' if you can execute this file (or cd to this directory)
5. file size in bytes

To get a string with any field use Field(File(*s_name*), *i_fieldNumber)* . To get the size, use Integer(Field(File(*s_name)*,5)).
Example:

```
if File("/opt/icm/icm.rst")=="- - - - 0" print "No such file"

if Field(File("PDB.tab"),2)!= "w"  print "can not write"

if ( Indexx( File("/home/bob/icm/") , "d ? w x *" ) ) then
   print "It is indeed a directory to which I can write"
endif
            # Here the Indexx function matched the pattern.

if ( Integer(Field(File(s_name),5)) < 10 ) return error "File is too sm
```

`File ( last )`
returns the file name of the last icm-shell script `called` by ICM. In scripts `File(last)` can be used for the Help section. See also: `Path ( last )`

`File` (*T_IndexTable* `database`)

returns the file name of the first source file indexed. Example:

```
 read index "nci"
       File( nci database)
/data/chem/nci.sdf
```

Find

function searching all fields (arrays) of a table, and to search patterns in sequences or their names.

Find closest value in array

`Find` (*R_source r_value*)

`Find` (*I_source i_value*)

- returns index of the *source* array element which is closest to the *value*

Example:

```
Find( {10 20 30 40 50} 43 ) #will return 4 because 40 is the closest val
Find( {1. 2. 3.} 100. ) #will return 3
```

See also: `Index`

Find text in tables.

`Find` (`table` *s_searchWords*)
- returns `table` containing the entries matching all the words given in the *s_searchWords* string.
If *s_searchWords* is "word1 word2" and table contains arrays a and b this "all text search" is equivalent to the expression :

```
(t.a=="word1" | t.b == "word1") & (t.a=="word2" | t.b == "word2").
```

Examples:

```
read database "ref.db"  # database of references
group table ref $s_out  # group created arrays into a table
show Find(ref,"energy profile") & ref.authors == "frishman"
```

`Find` (`table` *s_pattern* `regexp`)

- returns `table` containing the entries where at least one text column matches *s_pattern.*

Examples:

```
add column t { "one" "two" "three" } {"Item1", "Item2" "Item3" }
Find( t "Item[12]" regexp )  # matches first two rows
Find( t "two|three" regexp )  # matches last two rows
```

Find chemical substructures.

`Find(` *mol_array*, *array_of_chemical_patterns S_labels*)

`Find(` *mol_array*, *table_with_chemical_patterns*)

returns a 'sarray of chemical-pattern labels found in the *mol_array.*

If the **table** argument is provided as the source of the chemical patterns, the function will look for two columns:

- `.mol` array of chemical patterns
- a column called `".LABEL"` or `".LABELS"` in either upper or lower case.

The patterns can be specified using the wild cards permitted by the Molsoft chemical editor.

Example:

```
Find( chemTable.mol, Chemical( {"c1ccccc1", "[CH3]"}  ), {"benzene", "met
# or
group table t Chemical( {"c1ccccc1", "[CH3]"}  ) "mol" {"benzene", "methy
Find( chemTable.mol, t )
```

See also: `Index chemical` `Nof` `find table` `find molcart`

: Find(sequence)

returns an `sarray` of sequence names in which the sequence matched the pattern, e.g.

```
make sequence 10  # generates 10 random sequences
Find( "*A?[YH]*" sequence )
```

`Find(` `sequence` *s_seq_name_pattern*) searches the pattern in sequence **names** rather than sequences.

Floor

rounding function.
`Floor` (*r_real* [*r_base*])
- returns the largest `real` multiple of *r_base* not exceeding *r_real.*
`Floor` (*R_real* [*r_base*])
- returns the `rarray` of the largest multiples of *r_base* not exceeding components of the input array *R_real.*
Default *r_base*= `1.0` .
See also: `Ceil`( ).

Formula

`Formula(` *chemarray*)

- returns the `sarray` of compounds' molecular formulas.

Getarg

function returning the value for an argument to ICM or an `icm-script`. If one runs `icm` directly, specify arguments after the `-a` option,

e.g.

```
icm -s  -a t=2 verbose c='some text' # three arguments passed to icm
icm_script t=2 verbose c='some text' # three arguments passed to icm_scri
```

A summary of the Getarg functions:

- · : Getarg( )->
- · : Getarg(name)-> S_argNames
- · : Getarg(name [delete])-> S_files,e.g. '.icb'
- · : Getarg(set|list|mol|keep|sarray)-> S_argValues # mol or keep adds stdin and keep for chunk access
- · : Getarg([find|test]) ->