G42+ manual

J.C. Schön
Max-Plank-Institute for Solid State Research,
Heisenbergstr. 1, D-70569 Stuttgart, Germany.

7. Juni 2015

Zusammenfassung

Version: 7. June 2015

1 Introduction

G42+ is a computer program consisting of a number of modules that can be used to explore the energy landscape of a chemical system (rigid or flexible molecule / cluster, surface, periodic compound, or combinations thereof such as molecules on a surface or inside a periodic crystal). Three basic exploration methods are used: stochastic walkers, gradient descent, and the option of following a prescribed path. The dominant one is the use of stochastic walkers, which perform a Metropolis random walk.

For our explorations, we can use single walkers or many walkers in parallel at the same time. There are two reasons for introducing multiple walkers: A technical aspect is that on many modern computers it is possible to perform runs in parallel, and the most efficient parallelization consists usually in performing the same type of run for a single walker on each processor but with different random numbers in order to gain a sufficient statistics about the outcomes of the stochastic explorations. An algorithmic aspect is the fact that there exist a number of efficient schemes to explore the energy landscape that rely on the presence of many walkers and a permanent exchange of information among them, such as genetic algorithms, multi-walker repelling runs or multi-walker annealing schedules.

The landscape of the system that is explored is defined via three features: the state space, i.e. the individual atom configurations including the cell parameters of the periodically repeated simulation cell, the moveclass, i.e. the neighborhood of states that can be reached from a given state, and the energy function, i.e. the potential energy plus the pV-term for non-zero pressure and plus possible penalty terms forbidding the walker to enter some region of state space.

The elementary steps of the algorithm consist in generating or loading a configuration, followed by running a module. In fact, the program allows the generation of a meta-script, which contains thousands of commands that are used to perform many possibly complicated combinations of sub-runs using several different modules. In particular, one can use this metascript to repeat the same sequence of modules many times for different random numbers, which is important since the runs involve stochastic walks.

Parameters for atoms such as ionic radii as function of oxidation state are stored in a database. A second database lists all the so-called building groups, which are the central quantities that are used to describe an atom configuration during the exploration (the simplest building group is an individual atom, of course, while the more complex ones would be whole flexible molecules or pieces of surfaces). Finally, there exists a third database containing parameters of the so-called Gupta-potential.

The parameters for the individual runs are set in the file input.f, which is subsequently compiled together with the rest of the G42+-program. Array sizes are set within the file 'varcom.inc', which contains all the global variables. Note that all variables used within the program are either globally or purely locally defined. Exception are only the POWELL- and the simplex routine, which were taken from a book on algorithms. (CRYSTAL-variables??) The construction of the metascript is also done within input.f.

All the variables listed in input.f are subsequently saved in the protocol file of the current run. G42+ begins a new protocol file whenever a new configuration is generated or loaded.

There exist several files describing the atom configurations. In the basic type of a representation, various parameters such as the total number of atoms and building units are listed, followed by the three vectors describing the periodically repeated simulation cell. Next, the coordinates and the orientation of the various building groups are listed together with the coordinates of the atoms belonging to the building group relative to the center of the building group. Furthermore, the charges, magnetizations and radii of the atoms are given. Finally, the energy of the configuration plus the current values of the pressure and temperature are listed. Alternatively, the building units are not given and instead only the positions of the atoms inside the unit cell (fractional coordinates with respect to the three cell vectors) are listed. Similarly, we can list just the coordinates of the centers of the building units and drop the positions of the individual atoms. As a special case, if the study is only referring to atom configurations with a fixed symmetry, we can describe the configuration with the minimal number of (atom and cell) parameters needed. In this case, the basic type of representation is expanded and these free parameters are added.

Furthermore, we can define a background structure consisting of many (usually) fixed atoms in addition to the building units that are allowed to vary their positions and shapes. This structure is loaded in separately, and can be written out as part of the output files if desired, especially if the positions of the atoms in the background structure can change.

The type of atom configuration description is indicated by the prefix used in the filename. Furthermore, many prefixes and suffixes are employed to indicate, which module is currently active, which step of a more complicated run has been reached currently, and which random number seed is employed at the moment. Besides those prefixes and suffixes specific for a particular

module, we can have

1. 'start': indicates that we have generated or loaded in a new configuration to start a sequence of sub-runs with, and this configuration is saved as the 'start' configuration.
2. 'end': indicates that we have finished a sequence of sub-runs and are getting ready to load in or generate a new starting configuration. In that case, either the final configuration or the official output configuration of the last sub-run is saved as the 'end' configuration
3. 'inp': indicates the configuration that is used at the start of a module
4. 'outp': indicates the configuration that is the official output of the module
5. 'last': indicates the last configuration used or created by a module. This can be different from the official output of the module.
6. 'temp': indicates the configuration that is passed between two modules
7. 'im': indicates an intermediary configuration during a run. Usually it is used as starting point of some local exploration, and is subsequently used for the continuation of the actual exploration run
8. 'curr': indicates the current configuration of the system. This is used to check what is happening right now. One problem is that some machines keep this file in the buffer and the overwrite it all the time, such that we actually never get to see it!
9. 'traj': indicates a configuration along a trajectory on the landscape (saved because we want to follow the walker during the landscape exploration)
10. 'att': indicates a configuration that was attempted (and not necessarily accepted) during a threshold run
11. 'thdos': indicates the file containing density of states data from a threshold run
12. 'thatt': indicates the file containing energies of attempted moves from a threshold run
13. 'mcq': indicates local quench during Monte Carlo run
14. 'ptq': indicates local quench during parallel tempering run
15. 'loc': indicates the best configuration during a local run
16. 'imlr': indicates intermediary configuration during a salocrun

17. 'impabe': indicates the intermediary best configuration (in energy) along a prescribed path after local minimization from an intermediary point

18. 'impawo': indicates the intermediary worst configuration (in energy) along a prescribed path after local minimization from an intermediary point

19. 'sha': indicates a shadow configuration (created by minimization from a reference configuration (= a point along the exploration path, during e.g. a sa-local-run or a prescribed path)

20. 'ref': indicates the reference configuration from which a local minimization is started in e.g. a sa-local-run or a prescribed path.

21. 'beoutp': indicates the configuration with the best energy after finishing the path

22. 'wooutp': indicates the configuration with the worst energy after finishing the path

23. 'rw': indicates the configuration after a short local run with a free random walker

24. 'mc': indicates the configuration after a short local run with a constant temperature Monte Carlo

25. 'q': indicates the configuration after a short local run with a quench minimization

26. 'init5': indicates an intermediary configuration being saved while a init5-sub-routine is performed

27. 'mcbest': indicates locally best configuration during a Monte Carlo run

28. 'ptbest': indicates locally best configuration during a parallel tempering run

29. 'pressbest': indicates locally best configuration during a parallel tempering run for a given pressure

30. 'lbpre1': indicates locally best configuration found during a Monte Carlo run when local equilibrium has newly been reached

31. 'lbpre0': indicates locally best configuration found during a Monte Carlo run when local equilibrium has newly been lost

32. 'lbpre3': indicates a locally best configuration found during a Monte Carlo run when local equilibrium has newly been reached, that has been subsequently locally optimized

33. 'lbpre2': indicates a locally best configuration found during a Monte Carlo run when local equilibrium has newly been lost, that has been subsequently locally optimized

34. 'lbcur1': indicates current configuration during a Monte Carlo run while in equilibrium

35. 'lbcur0': indicates current configuration during a Monte Carlo run while out of equilibrium

36. 'lbcur3': indicates current configuration during a Monte Carlo run while in equilibrium which has been locally minimized

37. 'lbcur2': indicates current configuration during a Monte Carlo run while out of equilibrium which has been locally minimized

38. 'lbcur': indicates current configuration (during a parallel tempering run)

39. 'ptq': indicates a locally minimized current configuration during a parallel tempering run

40. 'jump' indicates a locally minimization after a jump move

41. 'mult' indicates a local minimization after a multi-walker move

42. 'M2' indicates a second local minimization during a call to the 'localrun' module

43. 'else' indicates a local minimization from an undefined module or during an undefined move

44. ADD MORE CASES

Quite generally, we usually have the number of the walker as a four-digit-character string at the end of the file name. Similarly, after the five-character that indicates the currently active module, we find a three-digit-character string indicating the number of the current module since a new set of subruns had been started (e.g. with its own 'confname'). For other naming conventions, see the section ... and the individual modules. (put all file naming conventions into one section???)

Each of these configuration files exist for every individual walker on the energy landscape, but the actual positions of the atoms and building units will be different for the various walkers, of course.

The basic modules that can be called are single energy calculation, simulated annealing, stochastic quench, threshold, shadow (or local)-run, thermal cycling (or multiquench), prescribed path, gradient descent, line search, POWELL, simplex, and findcell. Along the run, we can control the output,

i.e., how many details are to be written into the protocol, and how many intermediary configurations are to be written.

Most of these modules exist in three different variations: single walker (prefix 'one__'), multiple but separate non-interacting walkers (prefix 'mws__'), and multiple interacting walkers (prefix 'mwi__'). The multi-walker-separate modules are essentially the running in parallel or in sequence of many individual exploration runs, while the multi-walker interacting modules either contain moves that exchange configuration information between the walkers (e.g. cross-over moves like in a genetic algorithm), evaluate the acceptance of a stochastic move made by one walker based on the current locations of the other walkers, or adapt the acceptance criterion (i.e. usually the temperature) based on the current progress of all walkers.

Before calling a module, we can change the pressure, set a new (master) seed for the random number generator, change the current moveclass(es) (unless a particular value is required by the module), select a subset of walkers to be active, etc.

This program is both highly complex and very much alive: Many parts of it are under construction, revisions are being made continuously, and old (temporarily abandoned) parts still float around. Thus, not every option is active in every module, and warnings are given if one enters uncharted territories. Furthermore, while an attempt has been made to catch as many errors as possible which might be committed by the user in preparing the input.f file, the 'varcom.inc' file and the two required databases (atomdatabase and bgdatabase), and to generate appropriate error messages, it seems that every new user finds another way to confuse the code.

Thus, it is recommended to have some knowledge of Fortran programming, in order to be able to catch formatting errors, errors in picking the right array sizes, etc., or to be able to introduce write statements inside the code at appropriate places that would allow catching the non-obvious errors. In particular, when editing a file, make sure that all statements are within the columns 7 to 72. If you continue a line beyond column 72, then place a 'f' or '&' in column 6 of the next line. Else, only special line numbers (e.g. as targets for 'goto' commands) and the number of a format statement are allowed to be placed in columns 1 to 6 (if you do so, then begin this number in column 1). Note that throughout most of the G42+ code, I have started each command statement line at column 9 (or even further to the right), and whenever the content of a do-loop or of an if-then branching is entered, I move another two columns to the right, both for easier readability of the algorithmic structure. Of course, any information about bugs, errors and other problems encountered are greatly appreciated, and such problems (hopefully) will be eliminated in the next version of the code or will be flagged as official error messages to future users.

If you are going to add variables or other features to the code, please make sure that all variables shared by several subroutines are defined glo-

bally in the file 'varcom.inc', which contains both all global variables and the array sizes as global parameters. Except for counters in do-loops, use a prefix 'hv...' in all local variables you define. This way, you can more easily keep track of which variables in a subroutine are global and which are local. While it may not happen frequently, keep in mind that external codes are often making changes from one major release to another, and thus the input (and output !) files required (produced) might change from the ones assumed to exist at the time of the writing of the G42+ code! In particular, the sequence of entries in the files and/or the format of the input/output files can be different and would require changes both in the writeinput... files and the files where output from external codes is received and analyzed, in particular energy.f (for energy calculations), gd.f and localgraddesc.f (for local minimizations). In such a case, please contact me at some point to let me know what you are doing / have done, so I can update the 'official' version of the G42+ code, too.

2 Configuration files

The configuration files are formatted as given below. They include all information about the spatial arrangement of the atoms in the configuration. Within G42+, the basic unit is the so-called building unit, which can consist of one or more atoms or pseudo-atoms (large voids, vacancies, two-electron bonds, electron-filled spheres etc.). Thus, the file which serves as input and output for communication within or with the G42+-program must be the basic type. The other two types contain only the atom or building unit positions but not both. They are just part of the general output of G42+ in a format useful for future analysis. If we want to performs runs for fixed symmetry, we will use the so-called free-parameter representation, which describes a configuration by only those parameters (cell lengths, angles, fractional coordinates) that are allowed to be varied according to the symmetry of the configuration. These free parameters, together with the matrices needed to generate the complete structure from them, are added to the data in the basic type configuration file. The name of the basic type file is either given by the user (only for loading purposes, and needs to agree with the name in the general input, under which G42+ will load the file) or generated automatically by the program denoting the current state of the run. When generated by G42+, these files are characterized by the prefix 'no-'. The atom-only and building-unit-only file generated by G42+ has the same name as the basic-type file, with the characters at- and bg- added as prefixes, respectively, instead of the prefix 'no-'.

Before giving the detailed description of the configuration files employed by G42+, we note that in contrast to earlier versions of G42 the new G42+ employs flexible building units, i.e. the positions of the atoms in a building

unit with respect to its center can now change. But since this might not be desirable (or only desirable in certain limited ways, e.g. for rotations about some specific angles), the database contains, in addition to the usual pieces of information that are also included in the configuration file, information about which constraints in the molecule's configuration are to be enforced and which are free to change. These constraints are based on the bonding graph of the molecule, i.e. for each molecule we have information which atoms are nearest neighbors. Thus, the possible constraints are: fixed bond distances between neighbor atoms, fixed angles for triplets of neighbor atoms, and fixed dihedral angles for quartets of neighbor atoms. If we want to keep a larger subset of atoms rigid, e.g. a benzene-type ring, then we need to enforce the constraints on all the atom pairs, triplets and quartets that are involved in the formation of the ring. More details about this is given in the database section of the manual.

2.1 Basic type

2.1.1 List of 'read' commands in loading subroutine

In the following, we give a list of all the entries in the basic type configuration file, in form of 'read'-commands in a load subroutine, together with the required formats. Note that we do not read in information regarding the energy (and cost) of the configuration, or the pressure and temperature when it was generated.

- `read (10,90000='(a)') dumname`
  'dumname' is the root-name of the configurations during a given subrun (the run between two commands that load a new (external) configuration or generate a new configuration). When preparing an input configuration, one can either pick the official name which is being defined as part of the general input, or give it another name (which will be overwritten by the program and replaced by the official name).

- `read (10,90005='(2i2,i6)') dimens,crystflag,spgroup`
  'dimens' equals the dimension of the system; 'crystflag' equals one if we are dealing with a periodically repeated unit cell, and equals zero if the cell is not repeated (cluster!); 'spgroup' is the space group of the system (usually 1, unless we perform a run in the free-parameter representation)

- `read (10,90006='(i5,i3)') nbgcur, numbgtypes`
  'nbgcur' is the (current) number of building units in the system; 'numbgtypes' is the number of different building unit types used to describe the configuration. Note that one can employ the same building unit type several times, e.g. if one wants to keep one oxygen atom fixed while the others are allowed to move.

- `read (10,90010='(i5,i2,2i3)') natomcur, numtypes, chrgmin, chrgmax`
  'natomcur' is the (current) number of atoms in the system; 'numtypes' is the current number of atom types in the system. Again, each type can appear several times in principle, e.g. with different (fixed) charges; 'chrgmin' and 'chrgmax' are the lowest and highest charge allowed for the atoms in the system, respectively.

- `read (10,90015='(f7.1)') unitnum(i1)`
  This line is repeated i1 = 1,'numbgtypes' times. 'unitnum(i1)' is the number of the i'th type of building unit in the system.

- `read (10,90015='(f7.1)') elnum(i1)`
  This line is repeated i1 = 1,'numtypes' times. 'elnum(i1)' is the element number of the i'th type of atom in the system.

- `read (10,90025='(3f15.10)') ag(i1,1), ag(i1,2), ag(i1,3)`
  This line is repeated i1 = 1,3 times. 'ag(i1,i2)' is the component i2 of cell vector i1.

The next lines are repeated i1 = 1,'nbgcur' times.

- `read (10,90035='(3f15.10,2i3)') x(i1,1), x(i1,2), x(i1,3), bgtype(i1), bsize(i1)`
  'x(i1,i2)' is the coordinate of the center (of mass) of the building unit i1 with respect to cell vector i2; 'bgtype(i1)' is the type of building unit i1, 'bsize(i1)' is the number of (pseudo)-atoms in building unit i1.

- `read (10,90036='(3f10.5)') euler1(i1), euler2(i1), euler3(i1)`
  'euler1(i1)', 'euler2(i1)', 'euler3(i1)' are the three Euler angles of the building unit with respect to the underlying Cartesian coordinate system. The next three lines are repeated i2 = 1,'bsize(i1)' times.

  1. `read (10,90029='(3f15.10)') uref(i1,i2,1), uref(i1,i2,2), uref(i1,i2,3)`
     uref(i1,i2,i3) are the reference Cartesian coordinates i3 of atom i2 in building unit i1 with respect to the center of the building unit before we have applied the Euler angle rotation. It is necessary to keep track of uref during the calculation since for flexible building units (especially flexible molecules) the shape of the molecule can change in addition to its orientation and location as a whole.

  2. `read (10,90040='(3f15.10,i3,f10.5,i3)') u(i1,i2,1), u(i1,i2,2), u(i1,i2,3), atype(bgal(i1,i2)), charge(bgal(i1,i2)), ch(bgal(i1,i2))`

'u(i1,i2,1)', 'u(i1,i2,2)', 'u(i1,i2,3)' are the three components of the Cartesian-coordinate vectors from the center of building unit i1 to the atom i2 in the building unit after rotation by the Euler angles; 'atype(bgal(i1,i2))' is the type of atom i2 in building unit i1; 'charge(bgal(i1,i2)),ch(bgal(i1,i2))' are the charge (real number) and valence (integer) of atom i2 in unit i1, respectively.

3. `read (10,90041='(3f10.5)') rion(bgal(i1,i2,iload),iload), meff(bgal(i1,i2,iload),iload), magnetization(bgal(i1,i2,iload),iload)`
'rion(bgal(i1,i2))' is the radius of atom i2 in building unit i1 (including already multiplication by the radius scale factor belonging to the atom - for ways to drop this multiplication when loading in and using the rsfactor instead, see 'rsfactorflag'); 'meff(bgal(i1,i2))' is the effective mass of the atom i2 in unit i1 (including multiplication by the scale factor - this factor should be always one unless we are dealing with e.g. band electrons). 'magnetization(bgal(i1,i2))' is the magnetization of atom i2 in building unit i1.

Note that nearly all the variables have an additional index 'iload' (here only shown in the last line that is to be read in), which indicates that the configuration currently being loaded in will be assigned to the walker iload (typically, iload = 1,nwalk).

The next lines are added to the preceding ones in the basic-type configuration file when we vary only the free parameters (according to symmetry) instead of all atom and cell parameters. Note that this only works as long as the building units are only of size 1 (only one atom or vacancy etc.). First, we read in the atom parameters. The three components i3=1,3 of the position vector 'x(i1,i3)' for atom (=building unit) i1 are computed from the free parameters according to the formula: $x(i1, i3) = \sum_{i2=1,nfap} mfap(i1, i3, i2) * fap(i2) + bfap(i1, i3)$, where nfap is the total number of free atom parameters fap(i2) describing the configuration.

- `read (10,90050 = '(i7,i7)') natomcur,nfap`
'natomcur' = number of atoms in configuration; 'nfap' = number of free atom parameters needed.

The next lines are repeated i1 = 1,'natomcur' times.

- `read (10,90052 = '(3f10.5)') mfap(i1,1,i2), mfap(i1,2,i2), mfap(i1,3,i2)`
This line is repeated i2 = 1,'nfap' times. 'mfap(...,...,...)' is the matrix mapping the free parameters to the real atom coordinates.

- `read (10,90052 = '(3f10.5)') bfap(i1,1), bfap(i1,2), bfap(i1,3)`

'bfap(...,...)' is the constant vector that is added after the multiplication by 'mfap(...,...,...)'.

- `read (10,90051='(f10.5)') fap(i1)`
  This line is repeated i1 = 1,'nfap' times. 'fap(...)' are the nfap free atom parameters of the configuration due to symmetry.

Next, we add the free cell parameters. The six cell parameters are computed according to the formula $a/b/c = \sum_{i1=1,nLfap} TLfap(i2,i1) * Lfap(i1)$ and $\alpha/\beta/\gamma = \sum_{i1=1,nWfap} TWfap(i2,i1) * Wfap(i1) + BWfap(i2)$. From these the three cell vectors are constructed as follows: a along vector $\vec{a}_1$, vector $\vec{a}_2$ in (a,b)-plane, and vector $\vec{a}_3$ the rest (see moveclass for explicit formula).

- `read (10,90050='(i7,i7)') nLfap,nWfap`
  'nLfap' and 'nWfap' are the number of free cell lengths and angles, respectively.

- `read (10,90052='(3f10.5)') (Lparam(i1),i1 = 1,3)`
  'Lparam(...)' are the three resulting cell lengths

- `read (10,90052='(3f10.5)') (Wparam(i1),i1 = 1,3)`
  'Wparam(...)' are the three resulting cell angles.

- `read (10,90051='(f10.5)') Lfap(i1)`
  This line is repeated i1 = 1,'nLfap' times. 'Lfap(...)' are the free cell lengths.

- `read (10,90051='(f10.5)') Wfap(i1)`
  This line is repeated i1 = 1,'nWfap' times. 'Wfap(...)' are the free cell angles.

- `read (10,90052='(3f10.5)') (TLfap(i2,i1),i2 = 1,3)`
  This line is repeated i1 = 1,'nLfap' times. 'TLfap(...,...)' is the matrix for mapping free cell lengths to actual lengths.

- `read (10,90051='(f10.5)') TWfap(i1)`
  This line is repeated i1 = 1,'nWfap' times. 'TWfap(...,...)' is the matrix for mapping free cell angles to actual angles.

- `read (10,90052='(3f10.5)') (BWfap(i2),i2 = 1,3)`
  'BWfap(...)' are the constant angles that need to be added in the above matrix product.

The next three lines are not loaded in when G42+ reads the file, but should be entered for completeness (they appear as part of the output of G42+):

- `write (10,90090='(a4,e18.12)') 'E = ',E`

- `write (10,90090='(a4,e18.12)') 'p = ',press`
- `write (10,90090='(a4,e18.12)') 'T = ',T`
- If we use penalty terms during the calculations ('penaltyflag' = 1), then we also might want to register the cost (= energy + penalty). To to so, we set 'writecostflag' = 1 in input.f, and thus activate the write command
  `write (10,90040) 'C = ',cost`
- If we employ a mirror charge calculation for a surface, which is indicated by 'slabmirrorflag' = 0, then we have the write command
  `write (10,90040) 'Em= ',emirror`
- If we employ an idealized surface slab with an averaged van der Waals interactions, which is indicated by 'slabvdWflag' = 0, then we have the write command
  `write (10,90040) 'Ev= ',evdW`
- If 'plotflag6' = 1, then we register the value of the Ewald parameter for the configuration, and the 'best' value of this parameter for the run. This is usually not activated.
  `write(10,100) 'alpha = ',bta, ' alphabest = ',btabest`

2.1.2 List of formats

Below is the list of formats for the 'read' commands:

- `90001 format (2A)`
- `90000 format (A)`
- `90005 format (2I2,I6)`
- `90006 format (I5,I3)`
- `90010 format (I5,I2,2I3)`
- `90015 format(F7.1)`
- `200 format (A)`
- `90020 format (2F15.10)`
- `90025 format (3F15.10)`
- `90030 format (2F15.10,2I3)`
- `90031 format (F10.5)`

- `90032 format (2F15.10,I3,F10.5,I3,2F10.5)`
- `210 format(A,I5,A,I3,A,I3,A,I3)`
- `90035 format (3F15.10,2I3)`
- `90036 format (3F10.5)`
- `90040 format (3F15.10,I3,F10.5,I3,2F10.5)`
- `201 format (A,I7)`
- `90050 format (I7,I7)`
- `90052 format (3(F10.5))`
- `90051 format (F10.5)`
- `90090 format (A4,E18.12)`

2.2 Atoms only

2.2.1 List of 'write' commands in save subroutine

In the following, we give a list of all the entries in the atoms-only type configuration file, in form of 'write'-commands in a ouput (savecfg) subroutine, together with the required formats.

Note that nearly all the variables are the same as in the basic type file. Furthermore, essentially all variables that define a configuration will depend on the specific walker, of course.

- `write (10,90000) currfilename`
  'currfilename' is the current file-name G42+ assigns according to the command script.
- `write (10,90005) dimens,crystflag,spgroup`
- `write (10,90010) natomcur,numtypes,chargemin,chargemax`
  'natomcur' = current number of atoms; 'numtypes' = number of different atom types; 'chargemin' and 'chargemax' = minimal and maximal allowed charge in the configuration, respectively.
- `write (10,90015) elnum(i1)`
  This line is repeated i1 = 1,'numtypes' times. 'elnum'(i1) is the number of the atom type given in input.f.
- `write (10,90025) ag(i1,1),ag(i1,2),ag(i1,3)`
  This line is repeated i1 = 1,3 times, just as in the basic configuration.

- `write (10,90035) zrel(i1,1),zrel(i1,2),zrel(i1,3), atype(i1),ch(i1),rion(i1),magnetization(i1)`
  This line is repeated i1 = 1,'natomcur' times. 'zrel(i1,1)', 'zrel(i1,2)', 'zrel(i1,3)' are the fractional coordinates of atom i1 with respect to the three cell vectors. 'atype(i1)', 'ch(i1)', 'rion(i1)' are the atom type, charge (as integer, i.e. = valence) and radius of atom i1, respectively.

- `write (10,90040) 'E = ',E`
  'E' is the energy of the configuration

- `write (10,90040) 'p = ',press`
  'press' is the current pressure. Note that at the end of a sequence of sub-run G42+ often ends up writing the new pressure instead of the old one, if one has already updated the temperature and/or pressure before generating the new configuration.

- `write (10,90040) 'T = ',T`
  'T' is the current temperature. Note that at the end of a sequence of sub-run G42+ often ends up writing the new pressure instead of the old one, if one has already updated the temperature and/or pressure before generating the new configuration.

- If we use penalty terms during the calculations ('penaltyflag' = 1), then we also might want to register the cost (= energy + penalty). To to so, we set 'writecostflag' = 1 in input.f, and thus activate the write command
  `write (10,90040) 'C = ',cost`

- If we employ a mirror charge calculation for a surface, which is indicated by 'slabmirrorflag' = 0, then we have the write command
  `write (10,90040) 'Em= ',emirror`

- If we employ an idealized surface slab with an averaged van der Waals interactions, which is indicated by 'slabvdWflag' = 0, then we have the write command
  `write (10,90040) 'Ev= ',evdW`

- If 'plotflag6' = 1, then we register the value of the Ewald parameter for the configuration, and the 'best' value of this parameter for the run. This is usually not activated.
  `write(10,100) 'alpha = ',bta, ' alphabest = ',btabest`

2.2.2 List of formats

Below is the list of formats for the 'write' commands:

- `90000 format (A)`

- `90005 format (2I2,I6)`
- `90010 format (I5,I2,2I3)`
- `90015 format(F7.1)`
- `90025 format (3F15.10)`
- `90035 format (3F15.10,2I3,F10.5)`
- `90040 format (A4,E18.12)`

2.3 Building units-only type

2.3.1 List of 'write' commands in save subroutine

In the following, we give a list of all the entries in the building units-only type configuration file, in form of 'write'-commands in a output (savecfg) subroutine, together with the required formats.

Note that nearly all the variables are the same as in the basic type file.

- `write (10,90000) currfilename`
- `write (10,90005) dimens,crystflag,spgroup`
- `write (10,90010) nbgcur,numbgtypes,chargemin,chargemax`
  'nbgcur' = number of building units; 'numbgtypes' = number of different building unit types; 'chargemin' and 'chargemax' = minimal and maximal allowed charge in the configuration, respectively.
- `write (10,90015) unitnum(i1)`
  This line is repeated i1 = 1,'numbgtypes' times. 'unitnum'(i1) is the number of the building unit type given in input.f
- `write (10,90025) ag(i1,1),ag(i1,2),ag(i1,3)`
  This line is repeated i1 = 1,3 times.

For the next line, 'hvch' and 'hvrad' are set equal to dummy values 0 and 1.0

- `write (10,90035) x(i1,1),x(i1,2),x(i1,3),bgtype(i1),hvch,hvrad`
  This line is repeated i1 = 1,'nbgcur' times. 'x(i1,1)', 'x(i1,2)', 'x(i1,3)' are the fractional coordinates of the reference point (usually center of mass or an atom close to it) of the building unit i1; 'bgtype(i1)' is the type of building unit i1, 'hvch' and 'hvrad' are dummy values for the 'charge' and the 'radius' of the building unit i1. If necessary, these can be replaced by hand (or script) by physically appropriate values.
- `write (10,90045) 'E = ',E`

- `write (10,90045) 'p = ',press`
- `write (10,90045) 'T = ',T`
- If we use penalty terms during the calculations ('penaltyflag' = 1), then we also might want to register the cost (= energy + penalty). To to so, we set 'writecostflag' = 1 in input.f, and thus activate the write command
  `write (10,90040) 'C = ',cost`
- If we employ a mirror charge calculation for a surface, which is indicated by 'slabmirrorflag' = 0, then we have the write command
  `write (10,90040) 'Em= ',emirror`
- If we employ an idealized surface slab with an averaged van der Waals interactions, which is indicated by 'slabvdWflag' = 0, then we have the write command
  `write (10,90040) 'Ev= ',evdW`
- If 'plotflag6' = 1, then we register the value of the Ewald parameter for the configuration, and the 'best' value of this parameter for the run. This is usually not activated.
  `write(10,100) 'alpha = ',bta, ' alphabest = ',btabest`

2.3.2 List of formats

Below is the list of formats for the 'write' commands:

- `90000 format (A)`
- `90005 format (2I2,I6)`
- `90010 format (I5,I2,2I3)`
- `90015 format (F7.1)`
- `90025 format (3F15.10)`
- `90035 format (3F15.10,2I3,F10.5)`
- `90045 format (A4,E18.12)`

3 Energy calculation

Quite generally, the energy of a configuration can be computed in many different ways on many different levels of accuracy: via built-in empirical potentials, via (empirical) external potentials, via ab initio external potentials, or via penalty/constraint terms. Which level of accuracy and type of

energy function we want to use will depend on the kind of questions we ask at certain times of the exploration. Thus, we might want to perform some global exploration using a simple less accurate but fast-to-calculate energy function, and subsequently do some refinement employing very accurate but more time-consuming energy functions. Furthermore, we might want to create effective energy (or cost) functions by combining several different energy functions such as penalty terms, empirical potential and ab initio energy contributions. In particular, we might want to compute the energy interactions between some of the atoms using an empirical potential, while the interactions between others should be highly accurate.

In order to be able to do this, we have introduced for each atom-type involved a flag, which tells us, whether for a given moveclass this atom can participate in a specific energy calculation. These are the following: abinitatomtype(i,j,k), guptaatomtype(i,j,k), gulpatomtype(i,j,k), GROatomtype(i,j,k), AMBatomtype(i,j,k), FPatomtype(i,j,k), potatomtype(i,j,k). Setting this value equal to one indicates that atoms of type i may for moveclass j participate in the specific energy calculations for walker k.

Here, 'abinit' indicates ab initio calculations in general and needs to be specified in addition to specific ab initio energy types such as Quantum Espresso ('QE'). 'gupta' is the interaction using a gupta potential, which is an approximation to embedded atom potentials for metals. To use this, one needs to provide specific interaction parameters, either via an external database or directly inside input.f. 'gulp' refers to calculations using the GULP-program, where one needs to add an additional external library-file that contains the information about the parameters and potential type (the GULP code allows for many different types of interaction potentials between atoms). 'GRO' and 'AMB' indicate that the atom can participate in calculations using the GROMACS or the AMBER code, respectively. 'FP' refers to the ab initio program FPLO, and 'pot' allows the use of one of the in-built potential types. COMMENT: CURRENTLY ONLY ONE AB INITIO CODE CAN BE USED AT THE SAME TIME, AND THUS THE FP-ATOMTYPE ENTRY IS IGNORED

For the built-in potential types, the energy is usually calculated as the sum of four terms: a) a two-body potential term dealing with the pairwise interactions among the atoms (see below), b) a one-body term incorporating the ionization energy or the electron affinity associated with the formation of a cation or anion, c) a chemical potential term measuring the energy needed to add a neutral atom of a given type, and d) a pressure term p*(volume per unit cell), or an appropriate adaptation for clusters.

Currently, we are not activating the chemical potential term, since it has proven to be more efficient to keep the number and type of atoms constant for a given (sub)-run. The reason is that there is the general problem that if one uses the experimental chemical potential of e.g. an atom in a metal, this will end up favoring the atom to leave the simulation cell unless we are close

to the actual formation of the thermodynamically stable compound during e.g. a simulated annealing run. As a consequence one needs to set the moves that change the composition to zero

In the case of the Gupta-potential (for intermetallic compounds), the terms a) - c) are replaced by the multi-body Gupta-potential.

Similarly, in the case of the various types of external (ab initio or empirical) energy calculations, terms a) - c) are replaced by the energy computed by the external code. If necessary (perhaps due to low levels of accuracy as trade-off against speed), one may want to add a hard/soft sphere or Lennard-Jones term to keep atoms at an appropriate distance. This is one of the examples where it can be profitable, at least during the global exploration stage of a calculation, to employ both an ab initio method (or other external energy computation) and a built-in potential that either serves as a constraint / penalty type term or adds an empirical interaction that is not taken into account in the external energy function. One problem is that currently we cannot extract the gradient from the external potential; thus a gradient-minimization taking both the built-in empirical potential and the external energy function into account.

There are two special types of energy terms that have been added specifically in G42+: the interaction with idealized slabs, and the use of simple molecular modeling type constraint potentials for the description of the interactions inside a building unit.

The idealized slab represents an averaged out interaction of an atom / molecule / cluster with a perfect rigid infinite surface located at $z = 1/2$ in the periodically repeated simulation cell. In reality, a surface consists of many atoms arranged in a specific fashion (depending on what kind of surface we are dealing with). With the number of these atoms ranging into the thousands, computing the interaction with all atoms separately can be quite time-consuming. However, in many situations, the dominant contributions to the energy of the atom-surface interaction can be summarized under a) an average van-der-Waals type interaction represented by a Lennard-Jones potential term integrated over the whole surface, and b) an image charge interaction between the atoms and the dielectric (metallic) surface, according to their charges. In case a), each atom experiences an attractive interaction towards the slab that only depends on the distance to the slab and corresponds to a force acting orthogonal to the surface. In case b), we need to compute the interaction of all atoms with non-zero-charge with all image charges generated by these atoms. In both cases, we can scale the strength of these interactions by setting the interaction parameters explicitly.

For the case of the simple model interactions between the atoms inside a given flexible building unit, they essentially correspond to penalty terms where we can explicitly set the amount by which a parameter such as a bond-distance or am angle can vary from the idealized value it has according to the database. Furthermore, we can chose whether the penalty is a pure

step function (zero for allowed parameter values, and infinity for forbidden parameter regions) when trying to cross the limiting values or whether we have an increase in energy according to a parabolic-type potential until we reach the limiting values where the penalty function jumps to infinity.

In addition to energy terms and constraint/penalty terms that refer to a given configuration, there are penalty terms that are supposed to exclude a walker from a region of the energy landscape that has been explored before already. As will be discussed below, these can be static (based on earlier information about the landscape from previous runs) or dynamic. In the latter case, we are dealing with several walkers which create their own repelling field about them that keeps other walkers from getting close to them and thus keeps up the diversity of a many walker exploration instead of letting the walkers cluster in one or few basins.

There are some general flags that are set with regard to energy calculations ******REORGANIZE******:

- 'numbofcells' (must be odd integer) gives the number of basic cells in a row that are taken into account when computing the energy with a built-in potential: 'numbofcells' = 1: no periodic boundary conditions, i.e. we model a cluster; 'numbofcells' = 3: simulation cell is repeated once in all directions, i.e. we compute the energy of a 3×3×3 block of the repeated simulation cell, etc. Note that the information regarding 'cluster' or 'crystal' is carried over to calculations of the energy with external codes, at least if these codes can make a distinction between periodic or cluster calculations.
  `numbofcells = 3`

- If we are dealing with a cluster, we usually need to employ 'numbofcells' = 1. We can for local optimizations that are performed by external programs also require that the cluster is centralized afterwards, since these codes (especially GULP), sometimes place the atoms around the origin of the coordinate system instead of around the center of the simulation cell. We can take care of that by moving the cluster into the center of the simulation cell. For this we need to set 'centralizeflag' = 1. Note that if we deal with a background structure, we might not want to perform such a centralization.
  Also, QE and FPLO (and other external codes) offer to perform only local optimzations of atom positions. But for these, one usually does not move the atoms to the origin, and thus, we should not centralize the outcome of the minimization. Also, such atom position minimization might occur for periodic system with fixed unit cells!
  `centralizeflag = 0`

- 'cpflag' tells us, whether the chemical potential is always equal to 0.0 ('cpflag' = 0), equal to the full chemical potential ('cpflag' = 1), or

equal to a linear interpolation formula with respect to the melting point ('cpflag' = 2) or the boiling point ('cpflag' = 3). Usually, we employ no chemical potential as mentioned earlier.
`cpflag = 0`

- 'prevflag' activates the check of the reasonableness of the energy with respect to the number of repeated cells in the energy calculations: The calculation is repeated with a larger number of repeated cells and the energies are compared (when 'prevflag' = 1). Only makes sense for periodic systems and when using only internal potentials and very few repeated simulation cells such that cut-offs can play a role and favor non-sensical structures. Currently rarely used.
  `prevflag = 0`
  'tol7' is the value by which the energy is allowed to rise upon calculation with a bigger unit cell.
  `tol7 = 0.0d0`

- 'potflag' tells us, whether any built-in potential is supposed to be used at some point, either alone or in conjunction with an external code calculation: No potential: 'potflag' = 0; with potential: 'potflag' = 1. Remember that we need to set potatomtype to one (for a given moveclass), if we want to compute the built-in potential.
  `potflag = 1`

- 'pottype' is a flag that tells us, which built-in potential is supposed to be used. For more details on the available potentials see below.
  `pottype(1) = 2`

- 'abinitflag' indicates, whether an ab initio calculation of the energy is supposed to be performed: 'abinitflag' = 0: no ab initio calculation; 'abinitflag' = 1: CRYSTAL calculation; 'abinitflag' = 2: VASP calculation; 'abinitflag' 3: Quantum Expresso (QE) calculation; 'abinitflag' = 4: FPLO calculation. Note that this ab initio calculation is only performed for those atoms, for which 'abinitatomtype' is set to one for the currently used moveclass.

  When using CRYSTAL, one also need to edit the file 'inputcrystal', while for a VASP calculation, one needs to provide external files with information about the pseudo-potentials (pot.car etc.). The reason for this is that it is rather difficult to create the appropriate files inside G42+ just based on the input.f file. For QE and FPLO, this is possible, and one needs to fill the appropriate entries for the various parameters for the calculations in the QE and FPLO section (according to the help-pages of QE and FPLO). Quite generally, one should be familiar with the external code one is going to call from G42+, in order to be able to select the best choices of the parameters and/or

(pseudo)potentials.
`abinitflag = 0`

- 'mullflag' indicates, whether the charges should be updated according to the Mullikan analysis in CRYSTAL or a Bader-type analysis in VASP and QE: 'mullflag' = 1: yes; 'mullflag' = 0: no. Note that in the latter case, the Mullikan charges are automatically set equal to 0. This charge is needed, in order to estimate the proper ion-ion-distance when using hard/soft spheres as repulsive potentials to keep the atoms apart from each other. Also, this 'charge' can be used as input for an image charge computation when one performs an idealized slab interaction in addition to the energy calculation of the molecule alone.
`mullflag = 0`

- 'abinitinputflag' indicates, which of several possible ab initio inputs are chosen for calculations. The default is abinitinputflag = 1. This value can by changed explicitly be the command 'setabinitinputflag'. Currently, this option is not active.
`abinitinputflag = 1`

3.1 Simple built-in potentials

The type of the built-in potential is given by the value of 'pottype'. The simple potentials are mostly combinations of Lennard-Jones, and Coulomb (shielded or with Ewald summation according to DeLeeuw) terms, and soft/hard sphere terms:

- `pottype = 0`: No information specific to potentials is read in; only general atom information. Use this for gupta calculations that affect only part of the atoms present and where we have defined the gupta-parameters by hand, and where no other calculations with one of the built-in potentials take place.

- `pottype = 1`: Lennard-Jones

- `pottype = 2`: Coulomb plus Lennard-Jones (extended to include electrons, vacancies etc.)

- `pottype = 3`: Born-Mayer

- `pottype = 4`: Coulomb plus Lennard-Jones (eps depends on type of atoms and on charge state)

- `pottype = 5`: Coulomb + Lennard-Jones (Coulomb with DeLeeuw-method)

- `pottype = 6`: Coulomb + $r^{(-8)}$-repulsion

- `pottype = 7`: Coulomb + Hardsphere
- `pottype = 8`: Gupta-potential (no longer active: gupta caldulations are only controlled by guptaflag = 1)
- `pottype = 9`: only Hardsphere, with distances compared to sum of ionic radii (to be used together with ab initio calculations)
- `pottype = 10`: repulsive softsphere, with distances compared to sum of ionic radii (to be used together e.g. with ab initio calculations)
- `pottype = 11`: Specifically developed potential for the Si-B-N-system.
- `pottype = 12`: only hardsphere, with distances compared to pre-defined distances of atom-types i1 and i2 (in Angstrom), given in the array mindist(i1,i2).
- `pottype = 13`: Born-Mayer + short-range Lennard-Jones, and all within an overall distance cut-off with different parameters for each atom pair. Furthermore, the mindist-array is activated, i.e. atoms are not allowed closer than mindist(atype(n),atype(m)). For this, one needs to introduce the parameters of the various atom-pairings explicitly in input.f

Note that pottype can be different for different moveclasses i1. For example, we might want to perform a simulated annealing with only an empirical energy function (e.g. pottype(1) = 2), and then a quench with ab initio combined with a distance criterion (e.g. pottype(2) = 12).

Certain parameters are used in several potentials: 'damp' for the long-range cut-off of the Coulomb potential as an alternative to the deLeeuw-Ewald summation. Furthermore, there are fixed values of very-short distance cut-offs in the various potentials, besides the usual tolerances defined in input.f. For details see the implementation of the actual potentials below.

3.1.1 Lennard-Jones

Potential type 1 is pure Lenard-Jones. By itself, this potential is only used for neutral atoms. The entries for the arrays 'sig(...,...,...,...)' and 'eps(...,...,...,...)' contain the parameters of the Lennard-Jones potential, as function of atom type and atom charge, and are filled with information provided via 'atomdatabase'. Note that in this simple Lennard-Jones potential, 'eps(...,...,...,...)' does not depend on charge (assumed to be equal to zero). The implementation is:

```
if (R.gt.(0.18d0)) then
sgm = sig(atype(n),atype(m),ch(n),ch(m))/R
potential = (sgm**12 - sgm**6)*eps(atype(n),atype(m),0,0)
else
```

```
potential = 1.0d+13
nochange = 17
endif
```
R is the distance between atoms n and m.

3.1.2 Coulomb + Lennard Jones, with eps independent of atom types

Potential type 2 consists of a damped Coulomb potential plus a Lennard-Jones term, where we are using an atom-type independent value for epsilon, 'eps1'. Besides the charges of the atoms, the parameters needed for describing the potential are:

- The damping factor 'damp' gives the (exponential) long-distance cutoff for the Coulomb-potential.
  `damp = 0.10d0`

- 'eps1' is the (atom type and charge independent) epsilon value in the Lennard-Jones potential.
  `eps1 = 0.3d0`

- Finally, 'radiusscale' gives sigma = (rad1 + rad2)*'radiusscale', where 'rad1' and 'rad2' are radii of the two ions involved. Currently, this is not in use: we use 'rstype(...)' instead, which can be varied atom-specifically. Note that by varying the rstype-factor, we can essentially perform a computational alchemy that transforms one type of atoms into another as long as we only worry about the size of the atoms in the energy term (of course, the ionization energies etc. are different, but if the charges are kept constant, they do not influence the landscape exploration since they only shift the zero of the energy.
  `radiusscale = 1.0d0`

- 'tol1a' is the minimum distance allowed in the potential 2 (must be greater than 'tol2' below).
  `tol1a = 0.1d0`

- 'rsmin' and 'rsmax' are the extreme radiusfactors for the ions. If this is used, one more or less can ignore 'radiusscale'. Good for environment-dependent radii. These values are needed for calculations on landscapes where the ionic radius is part of the optimization, and can be adjusted within limits. Usually, this makes only sense if we are dealing with pseudoatoms such as electron clouds that can vary in size in response to Coulombic and Pauli-forces.
  ```
  rsmin = 0.1d0
  rsmax = 10.0d0
  ```

- 'potneutral' is the fraction of 'eps1', which is used to control the energy-repulsion of the neutral pseudoatoms. 'potneutral' only deals with the repulsive term: the 'in potentio' electron is trying to come into being far away from the other atoms.
  `potneutral = 0.1d0`

- 'chargepseudo' gives the maximum negative charge the pseudoatom is allowed to have.
  `chargepseudo = -10`

- 'potvacancy' is the potential associated with the vacancy-atom interaction (type Lennard-Jones). This does not have the restriction that the distance between atom and vacancy must be larger than 0.001. Remember that the atom/unit number of a vacancy is 150.0.
  `potvacancy = 0.0d0`

- 'distebond', 'potebond', 'angleebond', 'potangleebond' are variables that are used, if we introduce 'electrons available for bonding' (atom-number 950.0). The two body-term between bonding electrons is given by: $potebond * (-1 + 1/(distebond)^2)$. The three-body-term describing approximately the angle at the electron between the two atoms connected by the bond, is given by: $potangleebond * ((\cos(\theta) - angleebond)^2/(\cos(\theta) - 1.0d0)^2 - 1.0d0)$. The three-body term is activated, if 'ebondangleflag' is set equal to 1. Watch out, if the angle is smaller than 120 degree (three-center bond), since e.g. for 90 degree funny things might happen. The interaction between a bond-electron and a foreign atom goes via a Lennard-Jones term with strength 'eps1ebond'.
  ```
  potebond = 0.2d0
  distebond = 0.1d0
  angleebond = -1.0d0
  potangleebond = 2.0d0
  ebondangleflag = 1
  eps1ebond = 0.0d0
  ```

CHECK WHERE IN POT.F; ADD A CHECK FOR R ¿ 0.0001 IN POT.F
Implementation of potential 2:

```
if ((numelem(n).eq.150.0).or.(numelem(m).eq.150.0)) then
sumrad = radn + radm
sgm = sumrad/R
hv1 = sgm*sgm
hv2 = hv1*hv1*hv1
hv3 = hv2*hv2
potential = potvacancy*(hv3-hv2)*eps1
else
if (R.gt.(0.001d0)) then
```

```
sumrad = radn + radm
sgm = sumrad/R
hv1 = sgm*sgm
hv2 = hv1*hv1*hv1
hv3 = hv2*hv2
if (((numelem(n).gt.999.0d0).and.(chn.eq.0)).or.
((numelem(m).gt.999.0d0).and.(chm.eq.0))) then
potential = potneutral*hv3*eps1
else
potential = (hv3-hv2)*eps1 + 14.39d0*chargen*chargem*dexp(-damp*R)/R
endif
else
potential = 1.0d+13
nochange = 17
endif
endif
```

Note that 'nochange' = ... indicates in the output, why a particular move was rejected. (see below)

3.1.3 Born-Mayer

Potential type 3 is of Born-Mayer type. The entries in the parameter arrays 'sig(...,...,...,...)' and 'eps2(...,...,...,...)' must be provided via 'atomdatabase' (rarely used; usually defined explicitly with pottype = 13).

- 'rp' is the strength factor of the repulsive term
  `rp = 9.0d0`

The implementation is:

```
if (R.gt.(0.1d0)) then
sgm = sig(atype(n),atype(m),ch(n),ch(m))/R
potential = 14.39d0*ch(n)*ch(m)*dexp(-damp*R)/R + eps2(atype(n),atype(m),ch(n),ch(m)
else
potential = 1.0d+13
nochange = 17
endif
```

3.1.4 Coulomb+Lennard Jones with eps atom-dependent

Potential type 4 is a shielded Coulomb plus Lennard-Jones potential, with eps being atom-type dependent. The entries in the parameter arrays 'sig(...,...,...,...)' and 'eps(...,...,...,...)' must be provided via 'atomdatabase'. 'damp' has the same meaning as in potential type 2. (rarely used)
The implementation is:

```
if (R.gt.(0.1d0)) then
```

```
sgm = sig(atype(n),atype(m),ch(n),ch(m))/R
potential = (sgm**12 - sgm**6)*eps(atype(n),atype(m),ch(n),ch(m))
+ 14.39d0*ch(n)*ch(m)*dexp(-damp*R)/R else
potential = 1.0d+13
nochange = 17
endif
```

3.1.5 Coulomb with deLeeuw-method + Lennard Jones

Potential type 5 consists of a Coulomb term + a Lennard-Jones term, where the Coulomb potential is calculated by de Leeuw's method:

- The dipolterm that appears in the Ewald-summation for the built-in Coulomb-potential is added if 'dipolflag' = 1.
  `dipolflag = 0`
- The results of the Ewald summation are printed into the protocol-file if 'outpflag' = 1. Currently this option is not active.
  `outpflag = 0`
- The parameter 'jlimit' determines the number of cells that are taken into account in the calculation of the coulomb energy using an Ewald summation to be (2*'jlimit' + 1)**3.
  `jlimit = 1`
- The alpha-factor 'bta' in the exponentials of the Ewald-summation is found in an optimal way according the prescribed values of 'jlimit' and 'grenze0', and to activate the optimization of 'bta', we set `alphaoptflag = 1`.
- 'grenze' gives the accuracy for the minimizing routine alphamin.mw.f, which computes the optimal value of the factor alpha in the exponent of the Ewald-summation. The value of the accuracy 'grenze' is 10**'-grenze0'. A value of 'grenze0' = 4.0d0 is recommeded.
  `grenze0 = 4.0d0`

Regarding the implementation, only the Lennard-Jones term is listed, with epsilon, 'eps1', independent of atom type; the De-Leeuw term is in its own set of subroutines. 'sig(...,...,...,...)' is again provided via 'atomdatabase'.
The implementation is:

```
if (R.gt.(0.1d0)) then
sgm = sig(atype(n),atype(m),ch(n),ch(m))/R
potential = (sgm**12 - sgm**6)*eps1
else
potential = 1.0d+13
```

```
nochange = 17
endif
```

3.1.6 Coulomb+ r**(-8)-repulsion

Potential type 6 is a shielded Coulomb potential + a r**(-8)-repulsion term. (rarely used)
The implementation is:

```
if (R.gt.0.1d0) then
sgm = sig(atype(n),atype(m),ch(n),ch(m))/R
potential = 14.39d0*ch(n)*ch(m)*exp(-damp*R)/R + sgm**8
else
potential = 1.0d+13
nochange = 17
endif
```

3.1.7 Coulomb+ hard sphere repulsion

Potential type 7 corresponds to a Hardsphere- plus Coulomb-Potential using the deLeeuw-method for the Coulomb part. (rarely used) The following parameters must be set when generating a new configuration:
'initflag3' = 1; fact1 $\geq$ 1.0d0
This way an overlap of the atoms during initialisation is avoided.
Further parameters:

- 'tolhs' (%) is the tolerance for the ionic radii (radsum * 'tolhs')
  'tolhs' < 100 % –¿ ion is smaller; 'tolhs' > 100 % –¿ ion is bigger.
  `tolhs = 1.0d0`
- 'rad(...,...)' must be provided via 'atomdatabase'.

The implementation is:

```
sumrad = rad(atype(n),ch(n)) + rad(atype(m),ch(m))
if (R.gt.(sumrad*tolhs)) then
potential = 14.39d0*ch(n)*ch(m)*dexp(-damp*R)/R
else
potential = 1.0d+13
nochange = 17
endif
```

3.1.8 Gupta-potential

For the Gupta-potential, we used to set 'pottype' = 8, but now this is employed as an independent energy calculation controlled by 'guptaflag'; see subsection on Gupta potential.

3.1.9 Hardsphere repulsion

Potentials type 9 and type 10 are pure hardsphere and softsphere (repulsive) potentials, respectively. These potentials are supposed to be employed together with external code (especially ab initio) calculations, since they have only repulsive (R < (sum of ionic radii) * 'tolhs') values or are equal to zero (R > (sum of ionic radii)*tolhs). Hard sphere ('pottype' = 9) has a step function, Softsphere ('pottype' = 10) has a 'eps1' * r**(-12) potential, which equals zero for atom-atom distance ≥ (sum of ionic radii) * 'tolhs'. Remember to set 'initflag3' = 1 and 'fact1' ≥ 1.0d0, and choose an appropriate value for 'tolhs' for 'pottype' = 9, and to choose 'eps1' > 0 for 'pottype' = 10.

Specifically, although potential type 9 is only Hardsphere, in principle, one could also use it for Monte Carlo within a fixed cell to study amorphous systems. 'rion(...)' needs to be provided in 'atomdatabase'.
Implementation:

```
sumrad = rion(n) + rion(m)
if (R.gt.(sumrad*tolhs)) then
potential = 0.0d0
else
potential = 1.0d+13
nochange = 17
endif
```

3.1.10 soft sphere repulsion

Potential type 10 is repulsive softsphere (c.f. hardsphere repulsion). 'rion(...)' needs to be provided in 'atomdatabase'.
Implementation:

```
sumrad = rion(n) + rion(m)
hv4 = sumrad*tolhs
if (R.gt.hv4) then
potential = 0.0d0
else
sgm = hv4/R
hv1 = sgm*sgm*sgm
hv2 = hv1*hv1
hv3 = hv2*hv2
potential = eps1*(hv3-1.0d0)
endif
```

3.1.11 Si-B-N potential

Potential type 11 is a potential specifically developed for the Si-B-N system where Si and B are cations and N are anions. For more details, see the

SiBN.inc file.

3.1.12 Hardsphere (pre-defined distances

Potential type 12 is again a hard sphere potential, but with fixed atom-atom distances given by the array 'mindist(i1,i2)' for atom of types i1 and i2. This kind of potential is meant to be a supplementary potential for energy calculations with external codes where we want to avoid having to call the external code if two atoms are too close to each other.

3.1.13 Born-Mayer with specific parameters

Potential type 13 is a specific internal potential that includes more parameters plus a cut-off function for the Born-Mayer type potential analogous to the form commonly employed in many empirical potential simulations for ionic systems. The parameters are not loaded in but included in input.f. For details on the parameters, see the input.f file. Usually used for systems where fitted parameters are available in the literature.

3.1.14 Calculations with free electrons

Next, we have the variables associated with calculations of free electrons. Usually, they are used together with potential type 2 or 5. Free electron calculations are not active at the moment, but kept in the code for future reference. Remember to set 'electronflag' = 1, to indicate that electrons are present.

- 'lambda1' gives the fraction of Pauli-energy associated with the individual electrons, (1 - 'lambda1') the fraction associated with the whole electron gas outside of the ion cores ($0 \leq$ 'lambda1' ≤ 1)
  `lambda1 = 0.0d0`

- 'lambda0' gives the fraction of Coulomb-energy associated with the individual electrons, (1 - 'lambda0') the fraction associated with the whole electron gas outside of the ion cores ($0 \leq$ 'lambda0' ≤ 1)
  `lambda0 = 0.0d0`

- 'volfreeflag' (=1) indicates that the free volume is calculated and should be taken into account as a criterion. 'volfreeflag' = 1 is necessary, if one wants to use 'lambda0' or 'lambda1' less than 1.
  `volfreeflag = 0`

- 'coulomb0factor' multiplies the coulomb-self-energy of the electrons
  `coulomb0factor = 0.30d0`

- 'coulomb1factor' multiplies the coulomb-self-energy of all the electrons in the system assuming that they occupy a volume 'volfree' Usually,

one sets 'coulomb1factor' = 'coulomb0factor'.
`coulomb1factor = 0.3d0`

3.2 Gupta-potential (originally potential type 8 in old G42-code)

The general form of the Gupta-potential can be found in the literature (the implementation used in G42+ is given below). The potential parameters (Agup(i1,i2), Pgup(i1,i2), Rgup(i1,i2), Zgup(i1,i2), Qgup(i1,i2) for atoms of type i1 and i2) can be read-in from an external database, but it is probably better to include them directly in input.f (see section on 'guptadatabase' for information about the parameters).

Note that the Gupta-potential contains a term where the individual contributions are added and then the square root is taken (for details and motivation, see literature). Thus, certain conditions need to be fulfilled in the energy calculation: Only single atoms are allowed as building units, and only neutral atoms are allowed. The last condition is not critical, but it is helpful in order to make sure that the atomic radii make sense. However, if we introduce Rgup (the official distance-parameters of the atoms that are used in the gupta potential) explicitly inside input.f (to be recommended if one wants to use additional potentials to interact with the atoms), then we can use ionic radii for the other energy calculations without getting interference.

Within G42+, this is not a call to an external program; a Gupta potential exists within G42+. The parameters are either given here in input.f or in the read-in file in 'guptadatabase'. But in order to be able to combine the Gupta-calculations with other potentials, one should include the Gupta interactions as part of input.f.

Note that the Gupta-potential contains a term where the individual contributions are added and then the square root is taken. Thus, certain things need to be changed in the energy calculation: Only single (neutral) atoms are allowed as building units if they are involved in Gupta-calculations. Furthermore, for all atom types that have Gupta-interactions, we need to set 'guptaatomtype' = 1.

- 'guptaflag' = 1 activates the calculation of the Gupta-energy.
  `guptaflag = 0`
- The Gupta-parameters are read in from the 'guptadatabase' if 'readguptaflag' = 1.
  `readguptaflag = 0`
- If we want to read in the data from the 'guptadatabase', then we need to provide the number of how many atomtypes are involved in Gupta calculations, 'numguptatypes'. Furthermore, we need the element numbers 'elnumgupta'(1,...,'numguptatypes') of these atom ty-

pes, and which atomtype they belong to in the main list of atomtypes, 'atomtypegupta'(1,...,'numguptatypes') = e.g. (3,4,6).

```
numguptatypes = 1
elnumgupta(1) = 13.0d0
atomtypegupta(1) = 3
```

As an alternative, we include for those atoms that experience Gupta-type interactions the Gupta parameters directly in input.f.

- If 'guptasplineflag' = 1, we use the spline-cutoff version of the Gupta-potential. In that case, we need to add values for the two cutoff-points (i.e. end of standard Gupta-potential, and the end of spline-part at VGupta = 0) for each pair of atomtypes (make sure that rc(i,j) = rc(j,i))

```
guptasplineflag = 0
rc1(1,1) = 4.0d0
rc1(1,2) = 4.0d0
rc1(2,1) = 4.0d0
rc1(2,2) = 4.0d0
rc2(1,1) = 5.0d0
rc2(1,2) = 5.0d0
rc2(2,1) = 5.0d0
rc2(2,2) = 5.0d0
```

If we want to have only part of the atoms (e.g. the background structure or a cluster within/on top of a variable structure experience Gupta-interactions, and the other atoms use a different potential, then set 'guptaflag' = 1, and 'pottype' to the other type of potential. If atoms of the background should be included in the (Gupta)-calculations, set 'guptabackgroundflag' = 1.

Note that due to the complicated structure of the Gupta-potential (many-body-terms), we cannot separate mobile-mobile, mobile-background and background-background calculations; we have either only the mobile, only the background, or all atoms participating in the Gupta energy calculations, but nothing in-between. Of course, only types of atoms with 'guptaatomtype' = 1 will participate, so there is some freedom to play with if desired.

Note that if we want to have background atoms that can relax their atom positions, then we need to have 'fixflag' = 2 or 3. In that case, we must have 'guptabackgroundflag' = 1 or 2 if these atoms participate in Gupta-potential calculations. The case 'guptabackgroundflag' = 2 is most useful for the occasion when we are dealing with a metallic surface that is supposed to relax, but no metallic atoms are on top of it.

1. 'guptabackgroundflag' = 0: No background atoms involved;
2. 'guptabackgroundflag' = 1: Backgroundatoms+mobile atoms involved;

3. 'guptabackgroundflag' = 2: Only background atoms involved.

```
guptabackgroundflag = 0
```

The implementation of the Gupta potential is rather involved and involves summation over all (1+2c1max)(1+2c2max)(1+2c3max) neighboring cells:

```
eterm1 = 0.0d0
do n1 = 1,nbgcur(icalc)
hvgup1 = 0.0d0
hvgup2 = 0.0d0
do c1 = -c1max,c1max
c(1) = c1
do c2 = -c2max,c2max
c(2) = c2
do c3 = -c3max,c3max
c(3) = c3
do m1 = 1,nbgcur(icalc)
if ((n1.eq.m1).and.(c1.eq.0).and.(c2.eq.0).and.(c3.eq.0)) then
V2 = 0.0d0
eterm1 = eterm1 + 0.5d0*V2
else
do i2 = 1,dimens
hr(i2) = x(n1,i2,icalc) - (x(m1,i2,icalc)+c(i2))
c x refers now to BG, and not to individual atoms.
enddo
nmax = bsize(n1,icalc)
mmax = bsize(m1,icalc)
do n2 = 1,nmax
n = atype(bgal(n1,n2,icalc),icalc)
if (guptaatomtype(n,mclflag,icalc).eq.1) then
do m2 = 1,mmax
m = atype(bgal(m1,m2,icalc),icalc)
if (guptaatomtype(m,mclflag,icalc).eq.1) then
R = 0.0d0
do i3 = 1,dimens
h = 0.0d0
do i4 = 1,dimens
h = h + a(i3,i4,icalc)*hr(i4)
enddo
h = h+(u(n1,n2,i3,icalc)-u(m1,m2,i3,icalc))
R = R + h**2
enddo
R = dsqrt(R)
if (R.gt.tol2) then
```

```
hvA = Agup(n,m)
hvP = Pgup(n,m)
hvR = Rgup(n,m)
hvZ = Zgup(n,m)
hvQ = Qgup(n,m)
hvrc1 = rc1(n,m)
hvrc2 = rc2(n,m)
if (R.lt.hvrc1) then
hvgup3 = R/hvR - 1.0d0
hvgup4 = hvA*dexp(-hvP*hvgup3)
hvgup5 = hvZ*hvZ*dexp(-2.0d0*hvQ*hvgup3)
elseif ((R.ge.hvrc1).and.(R.lt.hvrc2)) then
hvpc1 = pc(n,m,1)
hvpc2 = pc(n,m,2)
hvpc3 = pc(n,m,3)
hvpc4 = pc(n,m,4)
hvpc5 = pc(n,m,5)
hvpc6 = pc(n,m,6)
hvgup3 = R - hvrc2
hvgup6 = hvgup3*hvgup3
hvgup7 = hvgup6*hvgup3
hvgup4 = (hvpc1 + hvpc2*hvgup3 + hvpc3*hvgup6)*hvgup7
hvgup5 = (hvpc4 + hvpc5*hvgup3 + hvpc6*hvgup6)*hvgup7
hvgup5 = hvgup5*hvgup5
else
hvgup4 = 0.0d0
hvgup5 = 0.0d0
endif
hvgup1 = hvgup1 + hvgup4
hvgup2 = hvgup2 + hvgup5
else
nochange(icalc) = 10
Ediff(icalc) = 2.0d1*Tinf
E(icalc) = Ecurrent(icalc) + Ediff(icalc)
goto 100
endif
endif
enddo
endif
enddo
endif
enddo
enddo
enddo
```

```
enddo
engup1(icalc,n1) = hvgup1
engup2(icalc,n1) = hvgup2
eterm1 = eterm1 + engup1(icalc,n1) - dsqrt(engup2(icalc,n1))
```

3.3 CRYSTAL

For calculations with the CRYSTAL code, we need to provide a sub-directory to the one where we perform our calculations, called 'Energycrystal'. G42+ will write the input-file for crystal into that directory, the script 'crystal-callkdl' will start the CRYSTAL run there, and we extract the energy computed from the CRYSTAL-outputfile generated in this directory.

Since it is not easy for the code to automatically generate some of files necessary for a crystal calculations, one needs to put the CRYSTAL input data that do not vary during the search into the file inputcrystal.f, in the form of an array of character strings, 'inpcrystal(1:linescrystmax)' (the maximum length of the array, 'linescrystmax', is set in 'varcom.inc'). You also have to set the value for 'numbofentries' in inputcrystal.f, which is the number of the last non-empty entry you are using in the array 'inpcrystal'.

For the meaning of the entries in the CRYSTAL-input, please refer to the CRYSTAL manual directly. Each entry consists of a character string, which is written into the 'inpcrystal' array. Of course, those entries which change from step to step (atom positions, cell parameters, etc.) are not listed here. They are generated in the subroutine writeinputkdl.

Note that the first five entries in 'inpcrystal' refer to the overall type of calculation that needs to be done, e.g.:

- `inpcrystal(1) = 'TEST'//confname`
- `inpcrystal(2) = 'MOLECULE'`
- `inpcrystal(3) = ''`
- `inpcrystal(4) = '1'`
- `inpcrystal(5) = 'END'`

The first entry is the name of the run where the calculation is done, while the second one ('MOLECULE' or 'CRYSTAL') tells us whether we are dealing with an energy calculation for a molecule or a crystal; the values for the third and fourth entry depend on whether we are dealing with molecules or crystals. At the beginning of the complete input-file that is read by the CRYSTAL code and generated in the subroutine writeconfkld, we write the entries one to four of 'inpcryst'. Next, G42+ writes cell parameters and atom positions, depending on whether we are going to do a molecule or a crystal

type calculations. The fifth entry should always be the information 'END', and will be written after the configuration information has been written.

The next entries in 'inpcryst' will be written in the order in which they are to appear in the CRYSTAL input file. They refer to basis sets, pseudopotentials, tolerances during calculations, type of calculation to be done, etc. For more details, see the CRYSTAL manual.

You also need to set up a script file before using CRYSTAL in G42 (crystalcallkdl) that calls CRYSTAL (wherever it is stored) and redirects the output appropriately, analogously to VASP and Quantum Espresso. A possible version of this script is:

```
#!/bin/csh -f
set INPUTFILE = $$
cp -f Energycrystal/INPUT Energycrystal/$INPUTFILE
./crystal < Energycrystal/$INPUTFILE > Energycrystal/$INPUTFILE.out
# for the energy we can easily grep
grep ENDED Energycrystal/$INPUTFILE.out > crystalenergykdl
if (-z crystalenergykdl) then
echo ' == no energy found E(AU) 1.0000000000000E+13 ' > crystalenergykdl
endif
#
grep ENDED Energycrystal/$INPUTFILE.out >> allenergies
grep 'T END ' Energycrystal/$INPUTFILE.out >> allenergies
#
rm -f Energycrystal/$INPUTFILE
rm -f Energycrystal/$INPUTFILE.out
```

For this, we provide the option **STILL NEED TO BE PROVIDED**** to either generate the script inside input.f or have it pre-prepared outside of G42+.

Note that we have not incorporated local minimizations using the CRYSTAL code.

3.4 VASP

For VASP, we need to provide an external folder called 'vasp', into which we place the files INCAR, KPOINTS and POTCAR. Furthermore, we need the program 'bader' that analyses the charge distribution file CHGCAR produced by VASP. You also need to set up a script file before using VASP in G42 (called: crystalcalvasp), such as:

```
#!/bin/csh -f
#set INPUTFILE = $$
set MPIDIR=/usr/local/server/intel/impi_3.2.2/bin64
cp -f INCAR Energyvasp/INCAR
cp -f KPOINTS Energyvasp/KPOINTS
cp -f POTCAR Energyvasp/POTCAR
```

```
#cp -f vasp Energyvasp/vasp
cp -f vasp528_MPI Energyvasp/vasp528_MPI
cp -f bader Energyvasp/bader
set num_tasks = `cat machine_list.* | wc -l`
cd Energyvasp
rm -rf vaspenergyaniket
rm -rf vaspoutput
rm -rf get_scf_number
#/raid/programme/theorie/VASP/vasp.5.2/vasp > stdout
#/opt/mpich/ch-p4/bin/mpirun -np $num_tasks -machinefile ./hostlist
vasp528_MPI > STDOUT
echo $num_tasks
$MPIDIR/mpirun -f ../mpd.hosts -r ssh -machinefile ../hostlist -np
$num_tasks ./vasp528_MPI > stdout
# bader charge analysis
#/home/kulkarni/bader/bader CHGCAR > outbca
bader CHGCAR > outbca
# for the energy we can easily grep
grep NELM OUTCAR >> get_scf_number
tail -2 OSZICAR >> vaspoutput
tail -1 OSZICAR >> vaspenergyaniket
tail -1 OSZICAR >> allenergies_vasp
cd ..
rm -f Energyvasp/POSCAR
rm -f Energyvasp/stdout
```

or

```
#!/bin/csh -f
# call with
# rung42 outputfilename
set TMPDIR = /scratch/kulkarni/$$
set CRYSDIR = /raid/programme/theorie/CRYSTAL/V2006_1.0.1/bin/Linux-pgf-opteron_P/v1
set MPIDIR=/usr/local/server/intel/impi_3.2.2/bin64
set VASPDIR=/home/c3theorie/VASP/bin
#set VASPDIR = /raid/programme/theorie/VASP/vasp.5.2/
set BADERDIR = /home/kulkarni/bader/
set here = `pwd`
set Aout = '/home/kulkarni/G42_GUCRYSVA_RS20april11_VASP'
set G42DIR = $here
set G42OUTPUT = $here/output
if (-e $G42OUTPUT) then
echo 'erase the old output directory first'
exit
endif
mkdir $G42OUTPUT
```

```
echo 'using CRYSTAL06, Linux version 1.0.2'
echo 'using VASP, Linux version 1.0.2'
echo 'using G42, directory' $here
echo cleaning, creating $TMPDIR
rm -rf $TMPDIR
mkdir -p $TMPDIR
if (-e restartinit) then
cp restartinit $TMPDIR
endif
#else if(abinitflag.eq.1)then
cp $Aout/a.out $TMPDIR
cp $CRYSDIR/Pcrystal $TMPDIR
cp $Aout/crystalcallkdl $TMPDIR
cp $Aout/crystalcalvasp $TMPDIR
cp externaldata $TMPDIR
cp $Aout/atomdatabase $TMPDIR
cp $Aout/bgdatabase $TMPDIR
#else if(abinitflag.eq.2) then
cp $Aout/a.out $TMPDIR
cp $VASPDIR/vasp528_MPI $TMPDIR
#cp $VASPDIR/vasp $TMPDIR
cp $Aout/vasp/INCAR $TMPDIR
cp $Aout/vasp/POTCAR $TMPDIR
cp $Aout/vasp/KPOINTS $TMPDIR
cp $BADERDIR/bader $TMPDIR
#else
#continue
#end
if (-e ../Limetal_ran_01/guptadatabase) then
cp $Aout/guptadatabase $TMPDIR
endif
# Achtung, naechster Befehl ist fuer etwaigen Restart
# Filename sollte auf .dat enden
echo 'hello 1'
cp $here/*.dat $TMPDIR
echo 'hello 2'
mkdir $TMPDIR/Energycrystal
mkdir $TMPDIR/Energyvasp
cd $TMPDIR
#cp INCAR $TMPDIR/Energyvasp/
#cp POTCAR $TMPDIR/Energyvasp/
#cp KPOINTS $TMPDIR/Energyvasp/
#
# create hostfile
```

```
#
/usr/local/fkf/bin/ll_get_machine_list_intelblade > machine_list.$$
set num_tasks = `cat machine_list.$$ | wc -l`
/usr/local/fkf/bin/strip_hosts < machine_list.$$ > hostlist
#if [[ $num_tasks -le 0 ]]
#then
# echo '$0: ll_get_machine_list is unable to create a machinefile.'
# echo '$0: Make sure that the binary ll_get_machine_list is accessible
to all
#machines in the cluster.'
# exit 1
#fi
#
echo ' ==> Total number of tasks: '$num_tasks
echo ' on nodes'
cat hostlist
echo `hostname` > mpd.hosts
setenv num_tasks
echo $HOST > $G42OUTPUT/dalaeuftes
echo $TMPDIR >> $G42OUTPUT/dalaeuftes
echo $1 >> $G42OUTPUT/dalaeuftes
date > $1.out
echo $HOST >> $1.out
nice a.out >>& $1.out
echo $HOST >> $1.out
date >> $1.out
echo 'hello 3'
rm -f $TMPDIR/a.out
rm -f $TMPDIR/Pcrystal
rm -f $TMPDIR/vasp
rm -f $TMPDIR/crystalcallkdl
rm -f $TMPDIR/crystalcalvasp
rm -f $TMPDIR/atomdatabase
rm -f $TMPDIR/bgdatabase
rm -f $TMPDIR/guptadatabase
rm -f $TMPDIR/fort.*
rm -f $TMPDIR/INCAR
rm -f $TMPDIR/POTCAR
rm -f $TMPDIR/KPOINTS
cd $here
echo 'hello 4'
#mv $TMPDIR/allenergies $G42OUTPUT/$1.allenergies
mv $TMPDIR/allenergies_vasp $G42OUTPUT/$1.allenergies_vasp
mv $TMPDIR/* $G42OUTPUT/
```

```
echo 'hello 5'
rm -rf $TMPDIR
```
for multi-processor runs of VASP (or CRYSTAL). Note that with appropriate modifications, the same script can be used for the script to perform runs with the CRYSTAL code for energy calculations. Nevertheless, you still need to separately provide the script crystalcallkdl for CRYSTAL calculations.

*****AS AN OPTION, THIS SCRIPT IS CONSTRUCTED IN INPUT.F - STILL NEEDS TO BE PROVIDED*** For details of what needs to be included in the files INCAR, KPOINTS and POTCAR, please see the VASP manual. The information about the current configuration is written in the subroutine writeinputvasp as the file POSCAR.

Note that in VASP the calculations of the energy of molecules and clusters also proceed as if they were periodic since the VASP code is based on plane wave basis sets. Furthermore, we have not incorporated local minimizations using the VASP code.

If we want to use the 'mullflag = 1' option to assign ionic radii to the atoms according to their net charge, one needs to enter the actual valence charges for ionic radius calculation. This is useful only for the VASP code. Assign a value to 'vaspcharge' for different types of atoms; note that these VASP-charges are identical for all walkers (for example, for Ca with twenty electrons, we put vaspcharge = 2 since all inner eighteen electrons are subsumed in the pseudo-potential.)

- `vaspcharge(1)=1.0`

3.5 Quantum Espresso (QE)

In contrast to the VASP and CRYSTAL modules, all information that is needed for QE-calculations is contained in the QE-section of input.f. This includes the script crystalcalQE which is used to call the QE-code. Note that both energy calculations and minimizations (including cell parameters) can be called from G42+. But one needs to warn: such minimizations can often stop before the minimum has been reached, and it is advised that whatever structures are found after local minimizations using the QE code, we repeat the minimization after a small perturbation to make sure that we have stopped at a real minimum.

Basically, go through the input.f section and carefully fill in the parameters that are needed. Since this is quite extensive, we do not give it here in detail (**perhaps include later??******). Most variable names are the same as in the QE-manual, just with a prefix 'QE_'.

Quite generally, do check the help-manual of QE for further information. For extensions beyond what is provided in input.f in the current version of G42+, one will need to add variables (preferably called 'QE_.....') and appropriately extend the writeinputQE subroutine (and possibly the energy

and gradient routines where the output of QE is analyzed). If one does this, one should keep in mind that the new (global) variables should be globally defined in 'varcom.inc'; remember: each global variable needs to appear twice in 'varcom.inc': once in the variable definition, and once in a common block!

3.6 FPLO

Similar to the case of Quantum Espresso, all parameters needed for FPLO calculations plus the script crystalcalFP that calls the FPLO code are set directly in input.f. Note that both energy calculations and local minimizations are possible, but the local minimization only includes changes in atom positions, not in cell parameters. Both molecules and crystals can be treated with FPLO.

Again, go carefully through the input.f section for FPLO and select the appropriate values in the input (*****perhaps include later??******). As much as possible, all input variables have a prefix 'FP_' but otherwise agree more-or-less with their names in FPLO. Please check with the FPLO manual regarding the meaning of these variables. The input file for the pipe-mode of FPLO is generated in the subroutine writeinputFP. If you need to add further features / variables, please proceed analogously to the case of Quantum Espresso.

3.7 GULP

For the subroutine GULP, we need to prepare one external file, the pot_lib file, which contains the information about the potential being used: type of potential and parameters that belong to this potential. Such parameters can sometimes be found in the literature for well-known systems and potentials, but sometimes one needs to spend time to find 'good' parameters that work well for the system.

As is the case for the other external codes, we use various flags to tell G42+ what kind of operations GULP is supposed to perform. In addition to the flags, we prepare the unchanging parts of the GULP input file explicitly in input.f, and then piece them together with the changing parts (generated in libGULP.f) to the complete GULP input file. We do not use scripts as we do for most of the other external codes to call the GULP code, but do the call to GULP as a system call inside G42+.

- `GULPflag = 1` indicates that the GULP-program is to be used at some time during the calculations. Note that we need to provide a file containing potential parameters for GULP, called pot_GULP.lib.
- `GULPenergyflag = 1` indicates that the energy is to be calculated using GULP

- `GULPgradientflag = 1` indicates that the gradient is to be evaluated using GULP (note that this is not yet fully implemented and should not be used)

- 'GULPoptimizationflag' > 0 indicates that a local optimization is being performed using GULP. Note: For clusters, use only 'GULPoptimizationflag' $= 1$. Note that for the case of periodic boundary conditions, we have two options: optimize both atom coordinates and cell parameters ('GULPoptimizationflag' $= 1$) or only the atom coordinates ('GULPoptimizationflag' $= 2$). The latter case is of interest, if one knows somehow already the cell parameters e.g. from experiment. Currently, the program assumes that GULP does not change the cell (and thus the reference of the fractional coordinates) for case 2. If we want to perform local optimizations with symmetries being fixed, one should probably change localgraddesc etc.; thus this option is not possible - all optimizations take place in space group P1.

  Note that if you include the background structure in the energy calculations, it is usually problematic, to change the cell during the optimization, even if 'fixflag' > 1. If 'fixflag' $= 1$, the code will crash if you try to change the cell parameters, since the background structure is not allowed to be changed, which includes the cell parameters. ****MAKE SURE THAT CLUSTER OPTIMIZATIONS ARE STILL POSSIBLE??****

Note that at the outset, enerGULPflag and optimGULPflag must be set to the default values zero and are changed later during the run.

Note that one can also run an energy calculation using a potential implemented in G42 before the GULP energy calculation. This can be useful if one wants to make a pre-check, whether e.g. two atoms are too close (some of the typical potentials used in the literature are singular for atom-atom distances near zero). The same type of pre-check is often used for ab initio calculations to avoid regions with bad convergence.

Furthermore, for speeding up calculations, it is sometimes better to program the GULP potential directly into the pot.f subroutine (for simple non-periodic GULP calculations), and use GULP only for the local minimization routine, i.e. 'GULPenergyflag' $= 0$ and 'GULPoptimizationflag' $= 1$. To use a built-in potential instead of / in addition to the GULP calculations, set 'potflag' > 0. Remember that 'potflag' depends on the moveclass - thus for energy calculations during a stochastic walk with one moveclass (usually number one) we can use the built-in potential, and for the local minimization with a different moveclass (usually number two) we employ the GULP minimizer. Another case where 'potflag' > 0 is useful, is if one wants to compute an interaction with a (fixed) background structure using the built in potentials besides the interactions among the mobile atoms and building

units.

If we want to include the interaction with the background structure into the GULP calculation, we need to set 'GULPbackgroundflag' appropriately.

- 'GULPbackgroundflag' = 0: background not included in GULP (but in G42-potential calculation)
- 'GULPbackgroundflag' = 1: included only in GULP
- 'GULPbackgroundflag' = 2: included both in GULP and G42-potential calculation.

Note that G42-potential calculation only takes place, if 'fixflag' > 0! Similarly, the GULP calculation of the background structure can only take place, if 'fixflag' > 0; else the background structure is 'invisible'. If we want to compute the interaction between / among background atoms with GULP, make sure that pot_lib includes the potentials for the atoms of the background structure.

Concerning the GULP minimization, keep in mind that we only are allowed to move the atoms of the background structure, if fixflag = 2.

Also note that if you are using (perfect) slabs, there will be a problem with the GULP-minimization, since GULP does not take mirror charges or fixed surfaces into account******check whether there are options in GULP in this respect*******. Also, remember that you need to be consistent regarding gradient calculations - don't mix GULP-energies with G42-gradients (which might not exist!).

The generation of the file pieces for the GULP_PARAM file proceeds by appropriately editing the following piece in input.f:

```
if (GULPflag.eq.1) then
open (12,file='GULP_PARAM')
write (12,1001) 'lib pot_GULP.lib'
c write (12,1001) 'output drv GULP'
1001 format(A)
if (GULPenergyflag.eq.1) then
write (12,1001) 'accu 8'
write (12,1001) 'rspeed 1.0'
endif
if(GULPgradientflag.eq.1) then
continue
endif
if ((GULPoptimizationflag.eq.1).or.(GULPoptimizationflag.eq.2))
then
write(12,1001) 'xtol opt 5.0'
write(12,1001) 'gtol opt 5.0'
write(12,1001) 'ftol opt 5.0'
```

```
write(12,1001) 'switch rfo 0.01'
write (12,1001) 'maxcyc 10000'

   c write (12,1001) 'update 10'
c write (12,1001) 'ftol 5'
c write (12,1001) 'gtol 5'
c write (12,1001) 'xtol 5'
c write (12,1001) 'stepmx 1.0'
endif
close (12)
endif
```

If you want to keep track of all the GULP-outputs, set GULPprotflag = 1. The name of the file is prot_GULP. If you want to write the most important configuration files in one of the other formats that GULP offers, set 'GULPformatflag' = 1, and define the keyword 'GULPformat' (exactly five characters, include empty spaces if necessary), e.g. `GULPformat = 'xyz '`

Finally, 'shellflag' = 1 indicates that a potential with electronshells is being used (at the moment only applicable in GULP, thus one usually just uses `shellflag = 0`.

3.8 Molecular modeling codes

Typically, the molecular modeling codes employ different type names for atoms depending on their environment inside a molecule, in order to pick the right kind of interaction term / parameters for their description. To be able to deal with this multiplicity of interaction-atom-types, we provide a list of atominttypes available for these atoms and indicate in the bgdatabase, which type a given atom is supposed to belong to. The file 'atominttype.inc' contains a list of all atoms that need special names for purposes of 'interactions' defined in special force fields (GRO..., CHARMM..., etc.). This file associates with the interaction number of each atom in the bgdatabase a specific 5-character (at least for GROMACS calculations) letter+number sequence. This file needs to be included in input.f, via the line `include 'atominttype.inc'`.

If we now want to associate atoms with certain interaction types as needed for the GROMACS and AMBER codes, and perhaps others, then we need to set 'atomintflag' = 1. If so, then make sure that for all the atoms in bgdatabase we have an entry (an integer number with one decimal, i.e. 1.0 or 10.0, etc.) at the key word '<atominttype>'! Furthermore, make sure that the 'atominttypename', which is needed in the external program is assigned in 'atominttype.inc' to the number in the bgdatabase. Until that has happened, set 'atomintflag' always equal to 0. The default is

`atomintflag = 1`
One general problem one encounters is that various external databases use different naming conventions for the atoms in the molecules, and it is not always clear, which is supposed to be the 'atominteractiontype'. Also, for large molecules like proteins, the potentials employ various kinds of coarse-grained descriptions which are not directly reflected in the way such molecules are encoded in the current code. To deal with such systems, it is usually necessary to edit the bgdatabase file by hand and assign the appropriate fixed/not-fixed flags for the various bonds and angles involved.

Similarly, we often need information about which ïnternalresidue etc. the atom belongs to within the building group. If we want to read in this information, we set 'atomresidflag' = 1. Make sure that there is an entry (an integer with a decimal, i.e.. 1.0, 10.0, etc.) at the key word 'atomresidtype' in the database! Furthermore, we need to assign to each atom in such a residue-containing building group a number, which tells us, which residue (in sequence) the atom belongs to. For 'atomresidflag' = 0, we automatically assign the number 0 to 'bgatomresidtype'.

The (4-character) string that gives the name of the residue of type 'bgatomresidtype' is given in a list that is called 'atomresidtypename' and must be included in the file 'atominttype.inc'.

The full specification of the residue requires information whether it is in the alpha-, beta- etc. form. This information is given in the 'atomresidtypechain' list in 'atominttype.inc'. Furthermore, we list in 'atominttype.inc', how many atoms are contained inside each of these (well-defined!) residue-subgroups. The list name here is 'atomresidtypesize'. Note that we are not restricted to peptides - we could also have a sequence of sugar-groups, etc..

For regular atoms in a non-residue-containing molecule in bgdatabase, the 'bgatomresidtype' number should be 1.0; 'bgatomresidnumber' = number of atom in the building group. Furthermore, we should give as entry 'atomresidtypename' the string 'atom', as entry of 'atomresidtypechain' the string '00', and as entry of atomresidtypesize 1, of course. `atomresidflag = 0`

3.8.1 GROMACS

The implementation of calls to GROMACS is a mixture of the way we call GULP and the external ab initio codes.

- 'GROflag' = 1 means we are going to call GROMACS
- 'GROenergyflag' = 1 means we perform single point energy calculations (should always be set to 1)
- 'GROoptimizationflag' = 1 means we are also going to call GROMACS for local minimizations (but it looks as if one can only optimize atom positions and not cell parameters?).

- If we want to optimize just atom positions, then we set 'GROvcrelaxflag' = 0, and if we also want to relax the cell, then we set 'GROvcrelaxflag' = 1.

- If we want to include background atoms into the GROMACS calculation, we need to set 'GRObackgroundflag' = 1 (only GROMACS is used for the energy calculation) or 'GRObackgroundflag' = 2 (in addition to GROMACS the built-in energy functions are used).

'optimGROflag' and 'optimGROvcflag' are internal flags, which must be set initially to zero.

For GROMACS, one should prepare beforehand a 'molecule.itp' file for each of the molecules to be used. Since it makes sense to re-use such a file, one should add this name (GRO...itp) to the building unit name in input.f. Each molecule / piece of a molecule is always associated with a building unit in bgdatabasemaster. We do need the information in bgdatabase because this information is important for additional interactions defined via the potentials directly inside G42 and because the database information about the connectivity of the molecule is used in the moveclass. The path to the directory where the .itp files are stored is given by 'GRO_molpfad', and the name 'GROitpname' of the .itp file is given in the definition of the building units: `GRO_molpfad = './Molecule_itp_files/'`.

When constructing the .itp file and the corresponding entry in bgdatabasemaster, make sure that they are consistent, i.e., the order of the atoms should be the same, and the right pairs, triplets and quartets are put together. Furthermore, do check (by inspection) whether a given distance / angle / dihedral is allowed to vary or not (i.e. value of 'freepairflag', 'freetripletflag', 'freequartetflag' = 1 or 0, appropriately). G42+ does check whether there are contradictions in the choice of these values, but a pre-check is to be advised.

Next, we need to worry about the interaction potential. 'GRO_potpfad' gives the path to the desired potential, and 'GRO_potential' is the name of the actual potential file. Make sure that the molecules / building units that are part of the simulation have the correct atominttyp-numbers. Also, make sure that there is an .itp at the end of the potential file. For example, `GRO_potpfad = './Gromos53a6/'` and `GRO_potential = 'forcefield.itp'`.

Once the .itp files for the molecules and the force fields are ready, we can construct the .top file describing the topology plus the interactions among the atoms in the molecule. Note that we do not include yet the information, how many molecules are going to be used - this might change during the run (e.g. we might include / exclude the background structure), and we therefore need to generate the full .top file just before calling grompp in energy.f and gradient.f (localgraddesc.f).

For the remainder of the many variables that are being pieced together into the various input-files needed for GROMACS and the script used to

call GROMACS, please go carefully through the section in the input.f file. If you want to add more features, you would have to add more variables (use prefix 'GRO_ for the variables in front of the variable name found in the GROMACS manual), and also edit the libGRO.f file, where the GROMACS input files are generated. Having the GROMACS manual at hand should be helpful, as for all the other external codes.

3.8.2 AMBER

******ADD AFTER GETTING LATEST UPDATE FROM SRIDHAR*********

3.9 Magnetization

In order to be able to generate magnetic structures on the energy landscape, we introduce the magnetization of each atom i, 'magnetization(i,iwalk)'. This can vary between -1 and +1 (normalized!), while the total magnetization of the cell is the sum over all magnetizations in units of Bohr magnetons.

At the moment, we are using QE as the code to evaluate spin-polarized electron configurations. This means that for a given magnetic distribution in G42+, the QE code tries to optimize the spin density during the SCF-cycles. We can either read in the outcome by analyzing the projected spin-dos, or just use the average (from the final total magnetization). Probably it makes more sense to go with the spin projections.

In the 'atomdatabase', we register the maximal value ('magnetmax' = number of up-spins - down-spins) which can exist at a given atom. Note that we will allow the system to violate Hund's Rules, i.e., we take as the maximum the number of valence electrons that might participate, in order to be able to deal with magnetic fields?

So far, we have not implemented this kind of magnetic information about the atoms in the 'atomdatabasemaster'! But we do have information about the magnetization in the 'bgdatabasemaster'. We use the magnetization by setting 'magnetflag' > 0.

- For `magnetflag = 1`, we give each atom a normalized magnetization of $+0.1$.
- For `magnetflag = 2`, we use a random starting value of ± 0.1 to allow the system to do its own adjusting, without including the spin polarization explicitly into the moveclass, or keeping track of the projected spin density. In that case, we can also prescribe a total constrained magnetization for the system, which it is supposed to achieve. (see QE-section for these variables.)
- `magnetflag = 3`: the system assigns each atom i to its own QE_type (but still with the right pseuodpotential, according to 'atype(i)'), and the initial magnetization is taken from the value of 'magnetatom(i)'.

```
magnetflag = 0
```

3.10 Penalty terms

In addition to the energy/enthalpy, we also compute a cost function, which includes explicit penalty terms that forbid the walker to enter certain regions of the energy landscape (cost = energy + penalty). In contrast to the 'penalty' introduced by e.g. forbidding two atoms to come too close to each other (like an exaggerated repulsive potential), these terms refer to whole regions of configuration space, independent of the physical reasonableness of the configurations - these regions are just taboo (usually because they contain structures that have been observed in an earlier run already).

- The penalty function penalizes a walker, if he gets too close to a forbidden structure (Taboo-search). Several types of 'order parameters' and penalty functions are used. If a penalty function is to be added to the energy, set 'penaltyflag' = 1, else = 0. (Other values for 'penaltyflag' can lead to problems.) Furthermore, note that we can either restrict the search to the neighborhood within the taboo-zone, or exclude the taboo-zone (see penaltyfunctionflag', below).

  Do not use the 'gradient' or 'linesearch' modules with a penalty function, since no gradient of penalty functions has been implemented.

  Do not use penalties with the 'prescribed path' option - this is nonsense. Use only pen = (0,infinity)-function with 'Monte Carlo' or 'Threshold' runs - else the energy average or the energy limit become pointless, respectively.
  ```
  penaltyflag = 0
  ```

- The number of forbidden structures is 'Npenstruct' ($\leq$ 'Npenstructmax' in 'varcom.inc')
  ```
  Npenstruct = 0
  ```

- 'writedbstructflag' equals 1: write up the 'dbstruct'-entries for the forbidden structures. The 'dbstruct'-entries describe the location in penalty space of configurations, and allows us to compare them. E.g. when we compare their pair correlation functions bin-wise, the corresponding array is defined as 'dbstruct(1:'Npenstructmax', 1:'Npairtypemax', 1:'Ndistbinmax')' in 'varcom.inc'. For a comparison based on cell parameters alone, we use the seven arrays 'cellastruct(1:'Npenstructmax')', 'cellbstruct(1:'Npenstructmax')', 'cellcstruct(1:'Npenstructmax')', diagabstruct(1:'Npenstructmax')', diagacstruct(1:'Npenstructmax')', diagbcstruct(1:'Npenstructmax')', 'volstruct(1:'Npenstructmax')'.
  ```
  writedbstructflag = 1
  ```

- The names of the files containing the forbidden structures, in standard building-unit format, are in the list 'penstructlist(...)'. Note that the order of atom-types must be the same as in the sub-run being undertaken, else the wrong pairs will be listed and compared. In order to ensure compatibility with the penalties chosen for the current run, the structures are read in and the comparison-arrays created at the beginning.
  `penstructlist(1) = 'endcfgt09999-55.0.8.0-01-00001'`

- 'penaltyaddflag' = 1: We add the last endconfiguration to the forbidden structures. Make sure that the total number of forbidden structures cannot exceed 'Npenstructmax' (in 'varcom.inc'). 'penaltyaddflag' = 0: No adding.
  `penaltyaddflag = 1`

- 'penaltytypeflag' = 1 : cell lengths and diagonal lengths are used as order parameter; 'penaltytypeflag' = 2 : binned atom-atom distances (listed by atom-pair-type) are order parameter; 'penaltytypeflag' = 3 : cell lengths and diagonal lengths are used as order parameter, but normalized with volume.
  `penaltytypeflag = 1`

- If we want to use absolute values for differences in 'dbstruct' when calculating penalties instead of square differences, set 'penaltyabsflag' = 1.
  `penaltyabsflag = 0`

- 'Npairtype' is the number of atompairtypes (= 'numtypes' * ('numtypes' + 1)/2 $\leq$ 'Npairtypemax' (in 'varcom.inc')) used for the binning.
  `Npairtype = 3`

- The width of a bin is 'distbinsize', the maximal distance in the pair correlation function is 'Distbinmax', the number of bins is 'Ndistbin' . This should be at least 'Distbinmax' / 'distbinsize' + 2 (because of leakage into neighboring bins, see below), and should be $\leq$ 'Ndistbinmax', (in 'varcom.inc'); distances are in Angstrom.
  ```
  distbinsize = 0.2d0
  Distbinmax = 8.0d0
  Ndistbin = 42
  ```

- Filling of the bins can be done with leakage into neighboring bins. The central bin where the distance lies receives 'sharp1', the bin to the left 'sharp0' and the one to the right 'sharp2' units (in integers). All dbstruct-values are finally divided by sharptotal = 'sharp1' + 'sharp0' + 'sharp2', in order to normalize.
  `sharp1 = 10`

```
sharp0 = 0
sharp2 = 0
```

- The filling of the bins can occur with various types of normalization, given by 'normdbflag': 'normdbflag' = 0: No normalization ('factor' = 1); 'normdbflag' = 1: 'factor' = 4 * pi * r**2; 'normdbflag' = 2: 'factor' = 4 * pi * r**2 * 'distbinsize'; 'normdbflag' = 3: 'factor' = 4 * pi * r**2 * 'distbinsize' * ('natomcur' / 'vol').
```
normdbflag = 0
```

- 'penaltyfunctionflag' = 1: Gaussian with temperature-dependence, 'Apen' * exp(-pen / ('alphapen' * T)); 'penaltyfunctionflag' = 2: Gaussian without temperature-dependence 'Apen' * exp(-pen / ('alphapen')); 'penaltyfunctionflag' = 3: Infinite-step potential, i.e. forbidden within radius penaltydist (standard taboo-search); penaltyfunctionflag = 4: Infinite-step potential, but this time forbidden outside of radius penaltydist' (inverse taboo-search). Remember to start from inside the taboo-zone for that kind of search! (else all moves are rejected!).
The original value of penaltydist is 'penaltydistorig'
```
penaltyfunctionflag = 3
Apen = 1.0d0
alphapen = 1.0d0
penaltydistorig = 3.0d2
```

- We write rejected moves due to penalty, if 'penaltychangeflag' = 1.
```
penaltychangeflag = 0
```

- We can automatically adjust the value of 'penaltydist' by calculating the average distance 'hv1av' between the structures, and assign a fraction 'penaltyfract' to be the penalty distance: 'penaltydist' = 'hv1av' * 'penaltyfract' (Note that 'penaltyfract' ≥ 1 will lead to ever increasing forbidden zones.)
'autopenaltyflag' = 0: no adjustment; = 1: with adjustment; the case 'autopenaltyflag' = 2 means that the fraction is taken with respect to the minimum distance among the forbidden structures, 'penaltydist' = 'hv1min' * 'penaltyfract'. In order to avoid a clumping of structures with 'penaltydist' going to zero, there is 'penaltydistmin', which constitutes a lower bound on 'penaltydist'. Similarly, 'penaltydistmax' gives a maximum value for 'penaltydist'.
```
autopenaltyflag = 0
penaltydistmin = 3.0d2
penaltydistmax = 1.0d6
penaltyfract = 0.2d0
```

- We automatically measure the distance to the last configuration that was present at penaltyupdate by setting 'distlastflag' = 1. 'distlastflag'

= 2 means that up to 'Nstructmax' (in 'varcom.inc') last configurations can be included.
`distlastflag = 1`

- The number of last configurations is usually equal to zero at the beginning ('Nlaststruct' = 0).
  `Nlaststruct = 0`

- However, one can, in principle, already provide entries to the arrays 'cellalaststruct(...)' etc.
  ```
  cellalaststruct(1) = ...
  cellblaststruct(1) = ...
  cellclaststruct(1) = ...
  diagablaststruct(1) = ...
  diagaclaststruct(1) = ...
  diagbclaststruct(1) = ...
  vollaststruct(1) = ...
  ```
  etc., for 'penaltytypeflag' = 1, and analogously (rather large) arrays 'dblaststruct' for 'penaltytypeflag' = 2.

- 'numattempts' gives the number of attempts for generating an initial configuration that is not within the penalty zone of forbidden structures.
  `numattempts = 10000`

- We can allow the starting configuration (randomly generated or loaded from a file) to be within a penalty zone by setting 'penaltypermitflag' = 1. This automatically sets 'penaltypermit' = 1, if the configurations is generated within a forbidden zone. Once the forbidden zone has been left successfully, 'penaltypermit' is set to 0, and the zones remain forbidden.
  `penaltypermitflag = 1`

4 System/configuration definition

An important part of the input is the definition of the chemical system that is being explored. This includes all the information needed to construct an input configuration file: the number of atoms and building units and their types and charges etc. Next, there is the rule according to which the starting configuration is generated if the configuration is not externally prepared and loaded in (see section ... for this part). Furthermore, we need to set the global restrictions on allowed moves of the atoms and the allowed changes in the cell during the global exploration, and various tolerances need to be defined.

Quantities such as the dimensionality of the system (currently only three-dimensional calculations are activated) and the space group have been defi-

ned earlier (see section ...). Note that in the input.f file, there are a number of initializations of arrays (mostly to zero) - do not change these initializations!

Note that, in principle, the number of atoms and building units can vary during a run between a minimum and maximum value, although these moves are not active at the moment. Thus, we usually set the maximum and minimum number equal to the initial number.

Furthermore, there are flags and entries that refer to the calculation of the energy using free electrons and free electron pairs; again, these types of calculations are not activated at the moment; do not change these values.

Note that you can give each walker its own composition, charges etc.! Only the types of atoms and building units must be the same, but nor their number (as long as it stays within the limits 'nbgmax' and 'nbgmin' etc.). Just replace the do-loops by individual definitions. In principle, this applies to all definitions in input.f, where a do-loop over the walkers is employed. But one should carefully think about this before planning the exploration / optimization with such different configurations!

4.1 Number of participating atoms and building units

- 'numbgmax' is the maximal number of building groups (bg) to be present during a run (must be smaller than 'nbgmax' in 'varcom.inc')
  `numbgmax = 6`

- 'numbgmin' is the minimal number of building units allowed in the system
  `numbgmin = 6`

- 'numatommax' is the maximal number of atoms (including pseudo-atoms of all kinds) to be present during the run (must be smaller than 'natommax' in 'varcom.inc')
  `numatommax = 6`

- 'natommin' is the minimal number of atoms (including pseudo-atom of all kinds) allowed in the system
  `natommin = 6`

- 'nbginit(...)' is the initial number of building units in the system (must be between 'numbgmin' and 'numbgmax'). This variable is set equal at the beginning to the current number of building units present, 'nbgcur(...)'. It can be different for different walkers.
  'natominit(...)' is the initial number of atoms (including pseudo-atoms) present (must be between 'natommin' and 'numatommax'). This variable is set at the beginning equal to the current number of atoms present, 'natomcur(...)'. It can be different for different walkers.
  'nbgreal(...)' is the total number of real building units present. Currently, make sure that for calculations with external codes only building

units are used that consist either of only real atoms or only vacancies, electrons or holes (forbidden zones)! Furthermore, note that in the background structure, only real atoms are allowed.
'natomreal(...)' is the total number of real atoms present, in contrast to the number of electrons + atoms + vacancies, 'natomcur'.
Note that any activation of the moves that change the number of atoms / building units will require a redefinition of 'natomreal' to 'natomrealcur' etc.

```
do i1 = 0,nwalkmax
nbginit(i1) = 6
natominit(i1) = 6
nbgreal(i1) = 6
natomreal(i1) = 6
enddo
```

- 'electronflag' = 1 indicates that electrons are present. These can be either free electrons or electron (pairs) inside a bigger building unit.
  `electronflag = 0`

- If vacancies are to be present in the calculations, set 'vacancyflag' = 1.
  `vacancyflag = 0`

- Similarly, if forbidden zones are to be present in the system, then set 'holeflag' = 1.
  `holeflag = 0`

Note that when listing below the various 'bgtypes' and 'atomtypes' that are part of the configuration, make sure that vacancies, forbidden zones, and electrons/electron pairs always follow after (!) all the real atoms and building units. The same holds true for pre-prepared structures one loads in from the outside: List all the vacancies, holes and electrons/pairs after all the real building groups have been listed.

Furthermore, one should not use building groups that contain both (!) real atoms and electrons/pairs when one employs external codes to compute the energies, and never if one performs structure minimizations with the external code. The problem lies in assigning proper new positions to the electrons/pairs after the minimization. Also, do not use vacancies, holes or electrons/pairs in the pre-prepared background structures.

4.2 Atoms and building units

Next we come to the definition which types of atoms and building units are used, and how many of them and with which charges, masses, etc.

- 'numbgtypes' is the total number of bg-types (must be smaller than 'bgtypemax' in 'varcom.inc'). In addition, we need the total number of bg-types that consist of only real atoms, 'numbgtypesreal'. The number of building groups of a particular type is given in the array 'initnbgtype(...,...). Each walker can have a different distribution of atoms, in principle. Note that we list both the atom types and the building unit types by number, and not by name.

```
numbgtypes = 2
numbgtypesreal = 2
do i1 = 0,nwalkmax
initnbgtype(i1,1) = 2
initnbgtype(i1,2) = 4
enddo
```

- Building-unit names: The crucial entry which is correlated with the entries in the 'bgdatabase' is the 'unitnum(...)'. 'unit(...)' only provides a descriptive name.
  Note that we can associate the same atom type or building unit type with several positions on the list, e.g. building-unit type 1 and 3 can both refer to oxygen atoms. Usually, one does this, in order to have more freedom what the properties of these atoms have - we might want to fix their charges at different values or fix the position of one group of oxygen atoms, etc. In some instances, we need to introduce different version of the atom also in the 'atomdatabase' to match the different atom definitions in the 'bgdatabase'.
  'bgname' is the 'unitnum' without the decimal point but given as character entry with 5 characters (needed for e.g. use in GROMACS). Finally, 'GROitpname' gives the name of the .itp file (needed for GROMACS calculations; make sure that the name ends with .itp). At the moment we use 'GROitpname' = 'bgname'.itp, but we are free in the choice of name - just be consistent, and remember that 'unitnum' must be a floating point number with one decimal. (As long as we do not use more than 9999.9 different types of building units, we are okay.).
  The 'analogue' to 'GROitpname' for AMBER calculations is 'AMBtopname': currently we just set it equal to 'coords.prmtop' (name of a crucial file in AMBER calculations).

```
unit(1) = 'Ti-atom'
unitnum(1) = 22.0d0
bgname(1) = '00220'
GROitpname(1) = '00570.itp'
AMBtopname(1) = 'coords.prmtop'
unit(2) = 'O-atom'
unitnum(2) = 8.0d0
bgname(2) = '00080'
```

```
GROitpname(2) = '00080.itp'
AMBtopname(2) = 'coords.prmtop'
```
Remember to set 'electronflag', 'vacancyflag' and 'holeflag' to one, if you use free electrons, electron pairs, vacancies and forbidden zones! Also, building units/atoms containing these pseudo-atoms need to be listed at the end of the lists of bgtypes and atomtypes.

Note that the decision which building units can be moved and/or change their shape is made in the energy parameter section.

- 'numtypes' is the total number of atom types (must be smaller than 'typemax' in 'varcom.inc')
```
numtypes = 2
```

- 'initntype(...,i)' gives the number of atoms of type i for a given walker (including those that belong to building units).
```
do i1 = 0,nwalkmax
initntype(i1,1) = 2
initntype(i1,2) = 4
```

- 'element(...)' just gives the element names: Length of element name should be 2 characters (leave blanks if needed). 'elnum(...)' is the actual reference to the information in 'atomdatabase'. Again, recall that the same atom type can be defined several times. 'atomtypename' must be five characters long (leave blanks if needed); this is needed e.g. for GROMACS and AMBER calculations, because for the molecules computed with these codes, atoms need different names depending on their location within the molecule.
```
element(1) = 'Ti'
elnum(1) = 22.0d0
atomtypename(1) = ' Ti'
element(2) = ' O'
elnum(2) = 8.0d0
atomtypename(2) = ' O'
```

Next we indicate, whether a particular type of atom is allowed to participate in a certain type of energy calculation for a given walker and a specific moveclass. For this we have 'abinitiatomtype', 'guptaatomtype', 'gulpatomtype', GROatomtype', 'AMBatomtype', 'FPatomtype', and 'potatomtype'. Note that this only gives permission for the atoms to participate, but this does not activate the external energy calculation itself; for that we need to set 'abinitflag', 'GULPflag', etc.

Note that currently, it is assumed that all 'natomcur' and all 'natomcurfix' atoms of a given type, if applicable according to the flags and moveclass currently active, are participating in the external potential calculations (GULP, QE, CRYSTAL, etc.)! Having only a subset of e.g. the background

atoms be involved in external potential calculations is not implemented. Of course, this can be achieved by defining several appropriate atom types for the same kind of atom, one of which allows calculations with an external code and/or the internal energy computer while the other does not.

- 'abinitatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in ab initio energy calculations for walker k.
```
do i1 = 0,nwalkmax
abinitatomtype(1,1,i1) = 1
abinitatomtype(1,2,i1) = 1
abinitatomtype(2,1,i1) = 1
abinitatomtype(2,2,i1) = 1
enddo
```

- 'guptaatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in gupta energy calculations for walker k.
```
do i1 = 0,nwalkmax
guptaatomtype(1,1,i1) = 1
guptaatomtype(1,2,i1) = 1
guptaatomtype(2,1,i1) = 1
guptaatomtype(2,2,i1) = 1
enddo
```

- 'gulpatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in gulp energy calculations for walker k.
```
do i1 = 0,nwalkmax
gulpatomtype(1,1,i1) = 1
gulpatomtype(1,2,i1) = 1
gulpatomtype(2,1,i1) = 1
gulpatomtype(2,2,i1) = 1
enddo
```

- 'GROatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in GROMACS energy calculations for walker k.
```
do i1 = 0,nwalkmax
GROatomtype(1,1,i1) = 1
GROatomtype(1,2,i1) = 1
GROatomtype(2,1,i1) = 1
GROatomtype(2,2,i1) = 1
enddo
```

- 'AMBatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in AMBER energy calculations for walker k.
```
do i1 = 0,nwalkmax
AMBatomtype(1,1,i1) = 1
```

```
AMBatomtype(1,2,i1) = 1
AMBatomtype(2,1,i1) = 1
AMBatomtype(2,2,i1) = 1
enddo
```

- 'FPatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in FPLO energy calculations for walker k.
  ```
  do i1 = 0,nwalkmax
  FPatomtype(1,1,i1) = 1
  FPatomtype(1,2,i1) = 1
  FPatomtype(2,1,i1) = 1
  FPatomtype(2,2,i1) = 1
  enddo
  ```
  At the moment, we are only allowing one type of ab initio energy function at a time (controlled by 'abinitflag'); thus this entry is ignored in the current version of G42+.

- 'potatomtype(i,j,k)' = 1 indicates that atoms of type i may for moveclass j participate in regular potential energy calculations for walker k.
  ```
  do i1 = 0,nwalkmax
  potatomtype(1,1,i1) = 1
  potatomtype(1,2,i1) = 1
  potatomtype(2,1,i1) = 1
  potatomtype(2,2,i1) = 1
  enddo
  ```

We now continue with indicating the charges, masses, etc. associated with the atoms initially. Some of these entries such as the mass will never change, while, in principle, the radius-factor or the charges can be changed in the moveclass. Such changes in charges and radii are only allowed for building units that correspond to single atoms - for building units with more than one (pseudo-)atom we cannot change the radii and charges set in the bgdatabase. Note that charges can only be changed in integer steps (one needs to pay 'ionization energy' or 'electron affinity' when adding/removing an electron), thus we sometimes speak of valence instead of charge. This valence will often be different from the precise charge defined for the building unit, especially, if the same atom appears with different charges in one or several building units. Thus, for this situation, one usually sets the valence of these atoms to zero. For consistency reasons, one might want to define several atom types for the same kind of atom, if it appears with different charges in different building unit.

- 'initnctype(iwalk,i,j)' gives the initial number of atoms of type i that have a valence (charge) j for walker 'iwalk'. This number must be an

integer, corresponding in some respect more to a valence than to a charge. Note that charges are not checked explicitly in 'inpclcwrite' for consistency! All atoms, also the ones with valence zero, need to have their valence defined here and in 'chtype' below. Since 'chtype' overrides 'initnctype', we no longer have the option to define different initial charges for the same atom type. If that is desired, then one should define for the same kind of atom several different atom types.

```
do i1 = 0,nwalkmax
initnctype(i1,1,4) = 2
initnctype(i1,2,-2) = 4
enddo
```

- 'chtype(iwalk,i)' gives the (initial) valences of atoms of a given type i for walker 'iwalk'.

```
do i1 = 0,nwalkmax
chtype(i1,1) = 4
chtype(i1,2) = -2
enddo
```

- 'chargetype(iwalk,i)' gives the (initial) charges of atoms of a given type i for walker 'iwalk' as a real number. Avoid fractional charges, if one wants to change the charge of atoms as part of the optimization.Note that in moveclass the charges are automatically set equal to the valences if a move is attempted where charges are changed. Furthermore the charge changing option is only possible for single atoms, i.e., building units of size one. Thus, one should set valences equal to charges in the input, if one wishes to vary the charges later.

```
do i1 = 0,nwalkmax
chargetype(i1,1) = 4.0d0
chargetype(i1,2) = -2.0d0
enddo
```

- 'magnettype(i,iwalk)' give the initial values of the magnetization of atoms of type i for walker 'iwalk' (always normalized to lie between ± 1).

```
do i1 = 1,nwalkmax
magnettype(1,i1) = 0.0d0
magnettype(2,i1) = 0.0d0
enddo
```

- 'rstype(i,iwalk) gives the current scalefactor of the radius of an ion or an electron pseudo-atom of type i for walker 'iwalk'. This variable takes the fact into account that the ionic radius changes with environment. Thus, exploration-runs should be repeated for different values of 'rstype', in order to check whether one has explored the 'right' kind of

landscape for the system (very often one finds additional good structures for slightly different values of the ionic radius). Should only be allowed to vary as part of the optimization if free electron calculations are performed (else, there is no Fermi-pressure keeping the system from collapsing!).

```
do i1 = 0,nwalkmax
rstype(1,i1) = 1.1d0
rstype(2,i1) = 1.2d0
enddo
```

- Note that when loading in a new configuration with 'loadnewcfg', we usually adjust the radius scale 'rstype' to agree with the radius indicated in the configuration file. If you want to insist that irrespective of the radius given in the configuration file, the scale factor should correspond to 'rstype', then set 'rsfactorflag' = 1. Note that this only applies for the 'loadnewcfg' command, not for every time you load a configuration with loadbg.f for other reasons!

  However, we set 'rsfactorflag' implicitly always to one (effectively in loadbg.f), if we are performing an ab initio calculation, since there the radius of the atoms is continually adjusted automatically due to the current charge distributions of the structure (if the option 'mullflag' = 1 is selected and we compute effective ionic radii for the atoms based on the ab initio calculation). As a consequence, when loading in a saved structure, we are encountering an ionic radius that may not have anything to do with the 'standard' radius rad(...,...) assigned during the loading according to the nominal charges listed in the atomdatabase. As a result the scale factor can become unreasonably large, and the walker will be stuck because of seemingly gigantic overlapping radii of the atoms in an actually otherwise good configuration.

  ```
  rsfactorflag = 0
  ```

- 'mscale(i,iwalk)' gives the normalization factor of the mass of atoms of type i for walker 'iwalk', mainly introduced for free electrons. Note that in the calculations, the mass of electrons is always given in units of the electron mass, i.e., meff(electron) = mscale(electron). For ions, we use the value given in the atomdatabase, and thus we would not pick a value different from one. In practice, 'mscale' should only differ from 1 if we model systems containing free electrons. However, in future applications, one can imagine molecular dynamics simulations where one would introduce an effective mass of the atoms that differs from the true atomic mass due to taking into account the average interaction with the environment such as a solvent.

  ```
  do i1 = 0,nwalkmax
  ```

```
mscale(1,i1) = 1.0d0
mscale(2,i1) = 1.0d0
enddo
```

- Shells are needed for some potentials to express the polarizability of an atom (e.g. some of those implemented in GULP) for atoms of type i. If a shell is needed, then 'shelltype' is set to one. 'shelltype(i)' are identical for all walkers.
  ```
  shelltype(1) = 0
  shelltype(2) = 0
  ```

5 Slabs, surfaces, and background structures

One of the major changes in G42+ compared to the older G42-versions is the ability to introduce (idealized) slabs, surfaces and other background structures, and also allow restrictions in the movements of the atoms and building units such that we can study e.g. quasi-two-dimensional systems such as surface layers.

5.1 Slabs

There are two meanings to the term 'slab': We either speak of idealized or perfect slabs that only exhibit an (averaged) interaction with the mobile atoms and molecules above them. There are two types of such slabs that can exist at the same time, of course: Idealized 'vdW-slabs' which interact via an averaged Lennard-Jones-type interaction with the atoms above them, and so-called mirror-slabs, which represent the metallic property(of metal) surfaces due to the rearrangement of charges inside the metal to neutralize the effect of the external charge (distribution) in the ions and molecules of the mobile building units. Since such a rearrangement can be represented effectively by a superposition of image charges with respect to the external charge (distribution), we speak of a 'mirror slab' when generating such image charges. Note that these idealized or perfect slabs do not exhibit any granularity on the atomic scale - they are perfectly smooth planes in space.

The second meaning of 'slab' refers to the construction of the slab that is supposed to mimic the surface of a solid, by cutting out one or several layers of atoms of a periodic crystal along certain directions, and using this set of several hundred or thousand atoms as a (fixed) background structure. Note that only for certain highly periodic 3d-crystal structures and low-integer choices of surface directions (e.g. 100- or 111-planes) is it possible to generate a slab that is periodic in two dimensions with a reasonably small number of atoms that yield a 'smooth' surface.

When dealing with slabs, the first thing to note is that they only show periodicity in two directions, usually chosen as the x-y-plane. Furthermore,

we do not want the mobile atoms and building units (molecules) to cross to the other side of the slab by moving via the periodic boundary conditions from the top to the bottom of the periodically repeated slab. Similarly, the energy calculations should only take the periodicity in two dimensions into account. Usually one chooses a very large distance in the directions orthogonal to the slab (typically the z-direction), such that atoms do not interact in this direction with their periodic images, for all practical purposes. In particular for certain external codes, especially plane-wave based ab initio codes, it is necessary to choose a large simulation cell in the z-direction.

- The variable that controls the periodicity is 'slabflag'. `slabflag = 1` indicates that periodicity only exists along the first two cell vector directions. The(idealized or perfect) slab extends in the x-y-plane, i.e. it is coplanar with the first two cell vectors. Similarly, if we define the slab as a background structure of (fixed) atoms, we have to make sure that it is parallel to the x-y-plane. `slabflag = 2` indicates that periodicity only exists along the first cell vector direction, corresponding to an infinite string.

  In both cases, this refers to the energy calculations. Atoms can still move in a periodic fashion, i.e., when they assume values of e.g. $z > 1$, then they reappear in the cell close to $z = 0$.

  `slabflag = 0` is the usual choice for systems with periodicity in three dimensions.

Regarding the energy calculations, one needs to keep in mind that the perfect slabs (mirror and vdW) do not consist of individual atoms, and thus their energy interactions with the rest of the system is always only via the built-in slab-mirror and slab-vdW potentials. In particular, this kind of interaction is not included in energy calculations with an external code, but appears as an additional term in the energy. In principle, the mirror charge effect should be included in ab initio calculations, but the vdW-type interactions are often not included in ab initio calculations and need to be added as an additional empirical term.

Concerning the choice of cell, if one deals with slabs, the third cell vector is usually orthogonal to the first two cell vectors. Similarly, for a string, the 2nd and 3rd cell vectors are orthogonal to the first one. But this is not enforced automatically.

Furthermore, the system as a whole is still three-dimensional - only e.g. movement and interactions in the z-direction are restricted. Usually, the slab/string, or part of it, is supposed to be rigid; thus one would not use moves that change the cell parameters during the explorations. If for some reason one wants to vary the cell parameter (e.g. when trying to optimize a slab of atoms in the x-y-plane), make sure that only cell moves are employed that do only change the cell parameters without (!) also changing the values

of the fractional coordinates of the atoms, i.e. do not use move options 6 or 12 where atom positions are unchanged when changing the cell parameters (and thus can end up in the next cell after the move). Similarly, be careful in the choice of jump moves or multi-walker moves.

For practical purposes, one might want to prevent the periodic movement of the atoms in the non-repeated directions. To achieve this, we have 'reflectflag':

- `reflectflag = 1`: moves into the z > 1 and z < 0 range are automatically rejected
- `reflectflag = 2`: moves into both the y > 1 and y < 0 range and the z > 1 and z < 0 range are automatically rejected
- `reflectflag = 3`: All moves into the outside of the simulation cell are rejected.
- If we are dealing with slabs, we usually want to keep the atoms/building units above the surface of the slab. `reflectflag = 4`: moves into the z > 1 and z < 0.5 range are automatically rejected
- `reflectflag = 5`: moves into both the y > 1 and y < 0 range and the z > 1 and z < 0.5 range are automatically rejected
- `reflectflag = 6`: All moves into the outside of the simulation cell and below z = 0.5 are rejected.

The standard setting for movement in all directions is `reflectflag = 0`.

These flags plus 'numbofcells' can be combined in all kind of ways. Thus, if slabflag = 1, reflectflag = 1 and numbofcells = 3, then the energy interactions are restricted to the simulation cell along the z-direction, but extend in the x-y directions into the first set of neighbor cells, and atoms are not allowed to move periodically in the z-direction (but atoms can still move in a periodic fashion in the x-y-direction). Or combining 'numbofcells' = 1 with 'reflectflag' = 3 corresponds to a cluster inside a reflecting simulation cell.

Using 'slabmirrorflag' and 'slabvdWflag', we can introduce a perfect surface that only interacts with the atoms in the upper half (z > 0.5) of the simulation cell via mirrorcharges for metal surfaces (`slabmirrorflag = 1`) and/or via a perfect vdW-polarization of the surface (`slabvdWflag = 1`). As an alternative to a vdW-surface, one can define a purely repulsive slab using (`slabvdWflag = 2`).

Note that this requires that the surface is perfectly aligned along the z = 1/2 level of the (at least) monoclinic simulation cell (a(2,3) = a(1,3) = a(3,1) = a(3,2) = 0). Furthermore, one should use a big enough value for the height of the unit cell c = 'a(3,3)'. This also implies that one should

use a reasonably large value for 'volmaxfact'. If one generates the starting configuration of the atoms at random, they will all be in the upper half of the cell (z > 0.6, in order to leave enough room between the atoms and the slab).

As an additional option, we can define two mirror- and/or vdW-slabs, one at z = 1/2, and one at z = 1, by setting 'slabmirrorflag' = 3 and/or 'slabvdWflag' = 3 or 4 (for two vdW and purely repulsive slabs, respectively). Then the starting atoms are restricted between z = 0.6 and z < 0.9. In this case, the system will move between two parallel layers, which might of interest for determining the arrangement of large molecules inside a nanochannel. Note that now the distance between the two slabs is determined by the choice of the cell-height in z-direction 'a(3,3)'/2. Also note that there is in principle an infinite number of image charges that are generated by the two mirrors, so the current implementation with just two images (one for top and one for bottom slab) is just an approximation.

If no perfect slab of any kind is present we set `slabmirrorflag = 0` and `slabvdWflag = 0`.

When using perfect slabs, we add additional interaction terms of the atoms with the surface without having to provide separate atoms for the surface. If one wants to introduce the atoms that provide the vdW-forces separately, then one needs to make sure that they are rigidly placed at z = 1/2 (and z = 1); else the mirror calculation will produce nonsense, of course.

For the vdW-interaction, we need as additional information the density of atoms in the surface layer, 'sigmasurf' (in atoms / Ang^2), the epsilon-value 'epsvdW' (in eV) (representing the strength) of the interaction, and the radiusöf the surface 'layer atoms', 'RvdWslab' (in Angstrom). Usually, one would use a similar value for 'epsvdW' as for 'eps1'. (In the future one could extend this to include different interactions at the top and bottom slab for the case of two slabs.) For example, we might take `sigmasurf = 0.2d0`, `epsvdW = 0.5d0` and `RvdWslab = 2.0d0`.

Similarly, if we calculate the mirror energy for a metal surface, we can modify its strength by providing a relative 'dielectric constant', 'epsrel'. For atoms above an ideal metal in a vacuum, it should equal 1, but this might be changeable depending on the actual type of surface and the medium in which the mobile atoms moves above the surface, `potslabmirror + 14.39d0*chmirrorn*chmirrorm/(epsrel*R2)`.

Note that we use at the moment the values in the charge-array as mirrorcharges 'chmirrorn' = -charge(n). If we perform ab initio calculations, we can, in principle, use the computed charges ('chmull(n)' which are computed for 'mullflag' = 1) as mirrorcharges by setting `abinitmirrorflag = 1`. However, as mentioned above, if we include the metal surface via background atoms in the ab initio calculations, then the image charges should have been taken into account already, and only, possibly, a vdW-type term is missing. The standard values would be `epsrel = 1.0d0` and `abinitmirrorflag =`

0

Finally, we can also introduce a quasi-pressure towards the slab (analogous to the pressure on an isolated cluster that aims towards the center of the simulation cell) by setting 'slabpress' (in eV/Ang^3, i.e. 'slabpress' = 1.0 corresponds to ca. 160 GPa!). To activate this option, we set `slabpressflag = 1`. This will move the atoms out of the gas phase towards the surface where the vdW- and the image charge-forces become dominant as far as the atom-slab interaction is concerned. This quasi-pressure term becomes zero for building units within a distance 'dslbpress' (in Angstrom) from the slab (at z = 1/2). Be careful to choose a large enough value of 'dslbpress' such that the atoms in the surface slab are not under actual pressure. Note that if we have two slabs, we still feel the 'pressure' only towards the slab in the middle. Furthermore, if one uses GULP to compute a pressure contribution, then the slabpress-contribution is ignored for consistency reason. Some values might be `slabpressflag = 0`, `slabpress = 1.0d3` (a very high quasi-pressure!) and `dslbpress = 5.0d0` (only one, or at most two layers of atoms on the surface).

On a final note: if we employ a background structure (regardless whether as slab or as some other kind of external environment), then the atoms of the background structure do not interact with the perfect slabs.

5.2 Single molecule

In many applications, we are going to deal with a (large) single molecule as the only unit that will change its shape and position with respect to a background structure and/or perfect slab. In that case, we can determine directly, where the molecule is to be placed inside the simulation cell (usually in the center) and which Euler angles it is to have (usually zero). To do so, we set `singlemoleculeflag = 1`, and prescribe the position 'xsm' and the Euler angles 'euler1sm', 'euler2sm' and 'euler3sm'. Note that if we do not know which Euler angles are compatible with e.g. a background structure, we set `rotateinitflag = 1`, and a random orientiation of the molecule is selected. If we want to enforce the Euler angles 'euler1sm' etc., then we must set `rotateinitflag = 0`.

The reason for allowing this kind of placement is that we want to avoid having the single molecule extend over more than one simulation cell (as can happen for a randomly placed / oriented molecule), which can lead to problems if the potential only sees atoms within the simulation cell but not in its periodically repeated copies (e.g. when setting 'numbofcells' = 1)! Furthermore, we might want to place the molecule at a specific distance from a background structure or mirror/vdW slab.

In this context: make sure that the simulation cell is large enough so that the whole molecule fits in, by prescribing the initial cell in the 'ainit'-array! For example for a molecule that is to be placed in the center of the

simulation cell without any rotation (for all walkers), we would have:

```
   singlemoleculeflag = 0
do i1 = 0,nwalk
xsm(1,1,i1) = 0.5d0
xsm(1,2,i1) = 0.5d0
xsm(1,3,i1) = 0.5d0
euler1sm(1,i1) = 0.0d0
euler2sm(1,i1) = 0.0d0
euler3sm(1,i1) = 0.0d0
enddo
```

If we want to randomly distort the atom positions inside a building unit, set 'urefinitflag' = 1. The maximal shift is given by 'disturef' (Angstrom).

```
disturef = 0.1d0
urefinitflag = 0
```

5.3 Background structure

Besides adding a perfect slab, we can load in a full background structure consisting of many building groups / atoms of various types. The idea is that we might want to have a set of interesting background structures such as slabs, zeolites, etc. ready at hand that are to be loaded in to describe the kind of chemical environment the mobile atoms and building units are supposed to encounter, and which, usually, will not be changed at all or, at most, relax a little bit due to the presence of the mobile atoms.

- To use a background structure, we set `backgroundflag = 1`.
- The name of the background structure file is given by the variable 'backgroundflname'.

When initializing G42, we then load in this background structure. In order to make sure that the same simulation cell is used both for the movable atoms and the fixed background atoms, one needs to generate the movable atoms in the new structure by using a prescribed initial cell with the same cell parameters as the background structure. Furthermore, note that in the background structure, only real atoms and building units that consist of only real atoms, are allowed: no vacancies, forbidden zones, electron pairs etc. should be part of a background structure.

- If the background structure is not to be changed during the calculations set `fixflag = 1` (usual situation). In that case, we do not include the background-background interaction in the internal potential calculations, in order to save computation time.
- If we set `fixflag = 2`, then we want to allow the background atoms to be able to change positions, as if they were regular atoms, but only during local relaxations with external energy functions.

But keep in mind that this might contradict the calculations with the internal potential that is allowed for 'abinitbackgroundflag' = 2 or 'GULPbackgroundflag' = 2 or 'guptabackgroundflag' = 2. (It can still make sense, if one e.g. has used the interaction potential only to mimic some long-range vdW-interaction to pull the atoms closer to the surface.)

- If we want the background atoms to be mobile for all calculations, we need to include the background-background interaction in all energy calculations (only makes sense if 'fixbg' > -2 for the building units / atoms that are part of the background structure, of course). This is achieved by setting `fixflag = 3`.

- Finally, `fixflag = 4` is the same as fixflag = 3, but we also save the resulting configuration including the background atoms.

- Note that if `fixflag = 0`, the background structure is ignored in the energy calculation.

One needs to be very careful, if you allow cell-relaxations for 'fixflag' = 2 ('fixflag' = 1 forbids cell relaxations). The entries of these flags may be: `backgroundflag = 0`, `backgroundflname = 'background'` and `fixflag = 0`

Furthermore, do not use any moves that change the cell during the global exploration when using a background structure. The same caveats hold when loading in a structure consisting of mobile atoms / building units using the 'loadnewcfg' module, together with a background structure.

When constructing the structure file for the background structure and possible (initial) structures to be loaded, and when setting up the parameters in input.f, make sure that the types of the atoms and building units include both those from the background structure and from the movable atoms. As a consequence, there will usually be building unit types and atomtypes listed in the mobile atom file to which no atom is associated since these types only appear in the background structure (and conversely)! Remember that you can define the same atomtype as coming from different building units (add the additional building unit types into bgdatabase if necessary). Thus, if e.g. the same type of atom is part of the fixed background structure and the mobile part of the system, then define two building units that correspond to this atom, where one of the building units is part of the background structure and the other appears in the mobile part.

In particular, the 'fixbg'-settings for the building units that compose the background structure are important to set correctly. If the building unit has 'fixbg' = -2, then this will override any movement, even when applying external minimization routines (provided that the external minimization routine allows keeping individual atoms fixed). Also, including atoms with

'fixbg' = -2 must not be mixed with cell optimizations (since these implicitly rearrange all atoms). The code does not explicitly forbid such a type of cell relaxation with external energy functions, but if one does this kind of calculation, one should not re-insert the outcome into the G42-optimization run.

The variables 'nbgcur' etc. still refer only to the movable atoms; 'nbgcurfix' etc. refer to the atoms in the background structure. Values for 'nbgcurfix' and 'natomcurfix' are taken from the background file, but one must ensure that they are smaller than the values of 'nbgmaxfix' etc. in 'varcom.inc'.

For most practical purposes, the background structure appears in the energy calculation like an external potential that is felt by the movable atoms. It is present for all walkers, and extends over 'numbofcells' simulation cells.

Usually, there are no interactions computed among the atoms of the background structure (see 'fixflag' for exceptions). Note that for interactions with the background structure we can only use potentials built into G42. If we want to use other (external) energy functions to describe interactions with the background structure, we need to include the atoms of the background structure in the atom list for the ab initio calculations by setting 'abinitbackgroundflag' and 'fixflag' accordingly.

When mobile atoms are generated at random, they are kept a certain distance from the background atoms, analogous to the allowed overlap with other movable atoms. If a multilayered slab is used as a background structure, one should place the top layer at z = 1/2, and the rest below z = 1/2. As mentioned above, background slabs can be combined with the perfect slabs described above. Note that the background structure can be of any kind and shape, i.e. it is not restricted to slabs.

Using the 'setfixflag' command in the command array, we can switch from 'fixflag' = 1 to another value, e.g. 'fixflag' = 0. When 'fixflag' = 0, the system behaves as if the background structure had vanished. If one wants to reactivate the background structure, one has to ensure that the cell parameters have not changed in-between. If we use this command to set 'fixflag' = 2, then from this moment all atoms of the background structure will be included in relaxations when using external codes, and analogously for 'fixflag' = 3 or 4.

Note that if the background atoms can change their positions (and possibly even the cell can change - watch out for problems then!), then we want to write out the changed background structure under certain conditions: if 'fixflag' = 2 or 3, then the changes in the background structure are not interesting in themselves so much, so we only write out the atom-coordinates of the background in a file with the suffix 'background' in the 'at-' file format.

But if 'fixflag' = 4, we also write out the file with the background atoms in the standard ('no-') format, so we can use it in further calculations as an (improved / relaxed) background-input. We never write out a 'bg-only'

type file for the background structure. If we want to avoid such files being written even for 'fixflag' = 2, 3 or 4, then we set `backgroundoutputflag = 0`, but if we want to write such files, we set `backgroundoutputflag = 1`. Note that currently we are not producing backgroundoutput for 'mwi'-modules or the 'pt'- or 'mc'-modules because the number of giant files would become too large (recall that typically a background structure is rather large - that is why it can serve as a background!). The standard setting is 'backgroundoutputflag' = 0

5.4 Atom layers

A special type of calculation associated with surfaces is the computation of optimal arrangements of atoms in a single layer or a multi-layer setting. This set-up is actually part of the choice of generating initial configurations and appears there again. But since it is related to slab-type calculations, we are going to discuss it here, too.

Such layers are always assumed to be parallel to the x-y-plane, and are thus determined by the value of the z-coordinate of the atoms in this layer. If we want to place the atoms randomly in n_zlayers layers with the same z-value, we set `zinitflag = 1`, zinit(i1,i2) = desired value of layer i1 for walker i2. This is usually used in combination with 'move2dflag' = 1, i.e. if the atoms are only allowed to move in the x-y-plane (z = constant), and if only the cell parameters 'a(1,1)', 'a(1,2)', 'a(2,2)' and 'a(2,1)' are allowed to be changed. For this we would preferably use only monoclinic cells that have 'a(2,3)' = 'a(1,3)' = 'a(3,2)' = 'a(3,1)' = 0.

Starting cells can either be prescribed at the beginning ('volinitflag' = 1) in toto, or we can prescribe only the length in z-direction by giving 'L_z', and the two remaining cell vectors are generated using 'volfactor' restricted to n_zlayers planes.

Filling more than one layer can be tricky, since for 'move2dflag' = 1 no moves are allowed to move atoms between the layers, and thus the number of atoms within each layer are fixed. Note that since we can exchange atoms between layers, the types can be changed; if one wants to also change the number of atoms, one needs to provide lots of vacancies that can be exchanged for an atoms between layers. Making it possible for atoms to jump between layers without the help of a vacancy is currently only a future option. If one wants to work with several layers with definite numbers of atoms per layer, where atoms can only move within the assigned layer, one can always prepare such an initial configuration file by hand.

- If we use random assignment of atoms positions, then we randomly distribute all atoms over all layers for `zinitflag = 1`.
- If we set `zinitflag = 2`, then the total number of atoms per layer is set equal to 'numatoms_zinit(i1,0,i2)'.

- For `zinitflag = 3`, 'numatoms_zinit(i1,i3,i2)' gives the number of atoms of type i3 that are prescribed as initial number within layer i1 for walker i2.

Make sure that the number of atoms per type defined for the whole system in 'initntype' correctly takes the distribution of atoms over the layers into account! (Note that we have not distributed atoms according to their charge, only according to type.)

Furthermore, if you combine 'zinitflag' = 1 with prescribed slabs or a prescribed background structure, you need to make sure that the values of 'zinit' are compatible with the restrictions of atoms on slabs. In particular, keep in mind that randomly placed (!) atoms are supposed to be at $z > 0.6$ (or $0.6 < z < 0.9$ for two slabs). It might happen, that in such a case, one needs to change the value of 0.6 explicitly in the code for some situations (subroutine init2.f). Currently, the choice of 'zinit' overrides this requirement as long as 'zinit' is larger than 0.5 (and smaller than 1.0 for two slabs)! Furthermore, for this kind of situation, the rim of the unit cell is off limits (i.e. as if 'rimflag' were set to zero).

As mentioned above, if you want to perform a quasi-2d-optimisation, where the atoms are only to be moved within planes, but you do not have a background structure to worry about, and the cell parameters in the xy-plane can change and should be generated the atom positions at random at the beginning, then use 'volinitflag' = 0 and prescribe the desired (correct) value for the c-axis by 'L_z'. If we do not use 'zinitflag' > 0, but still want to have 'L_z' a fixed value > 0, then we must set `Lzflag = 1`. A possible setting of these variables might be

```
   zinitflag = 0
n_zlayers = 1
do i2 = 0,nwalkmax
zinit(1,i2) = 0.5d0
numatoms_zinit(1,0,i2) = 12
numatoms_zinit(1,1,i2) = 6
numatoms_zinit(1,2,i2) = 6
L_z(i2) = 1.6d1
enddo
Lzflag = 0
```

6 Flexible building units

6.1 Control of allowed changes in positions and shapes of molecules

6.1.1 Overall control: fixbg

For all building units, whether they are single atoms, rigid multi-atom building units or flexible multi-atom building units, 'fixbg' defines, whether they can change as a whole, i.e. with respect to shifts and exchanges or shape changes:

'fixbg(i1,i2,i3)' indicates, whether for walker i1 a building group i3 may be moved during moveclass i2:

- `fixbg = 0`: can move and can change shape
- `fixbg = -1`: can be second partner in an exchange with a building unit with 'fixbg' = 0, but must not change otherwise.
- `fixbg = -2`: must not be moved and must not be exchanged
- `fixbg = i > 0`: can be exchanged with other building units with either 'fixbg' = i or 'fixbg' = 0.

Depending on how many moveclasses are used, one needs to define possibly different values for 'fixbg'. It is given as property of certain building units, but at the same time it reflects the moveclass - the second entry refers to the 'moveclassnumber', the third to the type of building unit.

Note that fixbg < 0 is not compatible with changes of the origin of the cell and the cutting off of a slice of the cell ('initmflag6' and 'initmflag7'). Changes of the cell by rescaling are okay, but need to be treated carefully. The same applies for relaxations with external potentials. Thus, if possible, avoid combining 'fixbg' < 0 with cell changes; currently, the moveclasses are not explicitly checking whether the atoms involved have 'fixbg' < 0 before attempting a cell move.

If we want to change the shape of a building unit or molecule, then we need to have 'fixbg' = 0.

```
do i1 = 0,nwalkmax
fixbg(i1,1,1) = 0
fixbg(i1,2,1) = 0
fixbg(i1,1,2) = 0
fixbg(i1,2,2) = 0
enddo
```

6.1.2 Specific features of flexible building units

In order to be able to deal with complex (organic) molecules, which possess a clear neighborhood structure, one can make the building units flexible.

In principle, everything can be changed, but as far as neighbor distances, angle between triplets of atoms, and dihedral angles for quartets of atoms are concerned, often simple constraints can be established. Note that for each building unit type and each distance between atom neighbors in the molecule, we can define the tolerance differently, i.e. by selecting 'tol' = 1.0d-10, essentially no change in the bond etc. is allowed. Similarly, the strength of the harmonic penalty depends on the type of building unit and the pair/triplet/quartet of atoms involved.

All atoms in the building unit that are not interacting via covalent bonding, experience a steric exclusion according to the array 'bgmindist'. 'bgmindist(i,j)' gives minimal distances between atoms of types i and j, which are the same for all building units. If one wants to have different distances for atoms belonging to different building units, then one needs to define more atom types. Note that currently this array is the same for all moveclasses. Further note that this array should be symmetric.

```
bgmindist(1,1) = 0.5d0
bgmindist(1,2) = 1.5d0
bgmindist(2,1) = 1.5d0
bgmindist(2,2) = 2.0d0
```

'flexbgflag(i1,i2)' indicates to what degree a building unit of type i1 can be changed when using moveclass i2.

- `flexbgflag = 0`: no changes are allowed.
- `flexbgflag = 1`: all the constraints are rigid, but if there are some other changes feasible, the structure is allowed to change in that respect.
- `flexbgflag = 2`: the building unit of type i1 can be freely changed within the given constraints, for moveclass i2.
- `flexbgflag = 3`: no constraints are activated - this should usually only be used if the energy of the intramolecular forces are computed via an external code, which already takes constraint-like forces into account.

If one wants to allow change in some molecules and not in others of the same type, then one should define them as two different building unit types. Obviously, for different moveclasses i2, 'flexbgflag' can have different values.

```
flexbgflag(1,1) = 3
flexbgflag(1,2) = 3
flexbgflag(2,1) = 3
flexbgflag(2,2) = 3
```

In addition to fixing the molecules on an individual basis, we have the overall flexibility flag 'flexibilityflag(i1)' for a given type of building unit i1. This is needed, in order to avoid the need to include data on pairs,

triplets, etc. in building units that will be completely unchanged. Thus, if the molecule is allowed to be changed at least partially, then one should set `flexibilityflag(i1) = 1`, but if a building unit is to be treated as completely rigid, then we set `flexibilityflag(i1) = 0`.

Note that if the building unit is of size 1 or if 'flexibilityflag(i1)' = 0, then all values of 'flexbgflag' for this bgtype are set to zero automatically when loading in the data from bgdatabase. In that case, bgdatabase will not be searched for information about pairs, triplets or quartets, and this information need not be included in the database. Setting 'flexibilityflag(i1)' = 0 overrides any setting of flexbgflag(i1,i2).

```
flexibilityflag(1) = 0
flexibilityflag(2) = 0
```

We can also perform moves that correspond to the rotation of a connected set of atoms that lie between two axis-atoms of a building unit (c.f. moveclass no. 24 and and the analogous jumpmove). 'bgsetpairflag(i1)' sets the parameter, if we really want to do this for building unit i1:

- 'bgsetpairflag(i1)' = 1: yes, if size of building unit is larger than 'bgsetpairmin' (and at least larger than 4)
- 'bgsetpairflag(i1)' = 0: no

```
do i1 = 1,bgtypemax
bgsetpairflag(i1) = 1
enddo
bgsetpairmin = 5
```

6.2 Penalty / energy terms associated with molecular constraints

Three kinds of constraints are possible:

- '...flag' = 1: no penalty within the interval ±'bg...tol(i1,i2)' around the prescribed distance (angle) between the atoms of pair/triplet/quartet i2 in building unit of type i1, and infinite penalty outside the tolerance interval about the ideal distance / angle.
- '...flag' = 2: harmonic penalty
- '...flag' = 3: harmonic penalty inside the tolerance interval, and infinite one outside the interval.

Usually, one provides information about pairs, triplets and quartets in bgdatabase. In this fashion, we can specify, which angles and bond lengths etc. are variable during the exploration of the energy landscape. But one can restrict oneself to only the pairs, and derive the triplets and quartets from the pair graph inside G42+. In that case, all these bonds and angles will be free

to vary, but when we enter these data into the bgdatabase, we can specify which ones are to be kept constant.In that case, the angles and dihedrals are computed from the atom positions.

Note that in the 'converalltoall'-routine (external to G42+), we can take a molecule given by its atom coordinates, e.g. in the pdb-format, prescribe the atom-atom-distances typical for bonding distances in the molecule, generate the pair graph and subsequently the triplet and quartets, and produce an entry ready for the database.

All these triplets/quartets that G42+ can generate automatically for building units of type i1 are either free or not, depending on the value of 'gentripletflag(i1)' / 'genquartetflag(i1)':

- Setting = 0: no generation, regardless of whether triplets / quartets are provided
- Setting = 1: fill empty(!) list with free triplets / quartets
- Setting = 2: fill empty(!) list with non-free triplets / quartets
- Setting = 3: fill up incomplete(!) list with free triplets / quartets
- Setting = 4: fill up incomplete(!) list with non-free triplets / quartets

```
gentripletflag(1) = 0
gentripletflag(2) = 0
```

Note that already if only one triplet or quartet is provided for the molecule ('bgtripletsize' > 0 / 'bgquartetsize' > 0 in the database), we do not compute any triplets or quartets, respectively, for the settings 1 and 2 of 'gentripletflag(i1)' / 'genquartetflag(i1)'. If we want to fill up an incomplete list with free or non-free triplets/quartets, use the setting 3 and 4. (Do not mix incomplete lists with settings 1 and 2)

Note that without the list of pairs it is not possible to generate the graph, triplets, quartets, or use any of the moves for flexible building units. Similarly, unless one provides / generates the triplets and quartets, the corresponding moves in the moveclass cannot be executed and will be ignored. Warning: if the atom-chain is linear, then a dihedral cannot be defined. Currently, the code sets the value of this angle to zero. ***Since this can be a general problem in the calculations, it might be useful, to drop the dihedral angle?***

If we want to have the code compute constraint terms in the energy calculation, we need to set 'ebgflag' = 1. Note that these constraint terms are only applicable to mobile building units, not to background building units.

```
ebgflag = 0
```

Such constraint terms apply to bond distances, angles for triplets of neighboring atoms, and dihedral angles for quartets of neighboring atoms. In each case, the same options of constraints apply.

- 'potpairbgflag' refers to distances between neighbor atoms

  1. 'potpairbgflag' = 0: no special constraint pair-potential active
  2. 'potpairbgflag' = 1: constraint potential equals 0 for distance within tolerance interval ('bgpairdisttol') about ideal distance, and equals infinity else
  3. 'potpairbgflag' = 2: harmonic constraint potential for all distances (zero for distance = ideal distance, of course)
  4. 'potpairbgflag' = 3: harmonic constraint potential for distances within tolerance interval, equals infinity else

  `potpairbgflag = 0`
  The necessary parameters for the constraints are

  1. 'bgpairdisttol(i1,i2)' is the distance tolerance for atom pair number i2 in building unit i1 (applicable for 'potbgpairflag' = 1,3).
  2. 'Aharmpotbgpair(i1,i2)' is the strength of the harmonic potential for atom pair i2 in building unit i1 (applicable for flag = 2,3).

```
do i1 = 1,bgtypemax
do i2 = 1,bgpairmax
bgpairdisttol(i1,i2) = 0.1d0
Aharmpotbgpair(i1,i2) = 10.0d0
enddo
enddo
```

- 'pottripletbgflag' refers to angles between triplets of neighbor atoms.

  1. 'pottripletbgflag' = 0: no special constraint triplet-potential active
  2. 'pottripletbgflag' = 1: constraint potential = 0 for angle within tolerance interval ('bgtripletangletol') around ideal angle, = infinity else
  3. 'pottripletbgflag' = 2: harmonic constraint potential for all angles (zero for ideal angle, of course)
  4. 'pottripletbgflag' = 3: harmonic constraint potential within tolerance interval, = infinity else

  `pottripletbgflag = 0`
  The necessary parameters for the constraints are

  1. 'bgtripletangletol(i1,i2) is the angle tolerance in degrees for triplet i2 in building unit i1 (applicable for 'pottripletbgflag' = 1,3).

2. 'Bharmpotbgtriplet(i1,i2) is the strength of the harmonic potential for triplet i2 in building unit i1 (applicable for 'pottripletbgflag' = 2,3).

```
do i1 = 1,bgtypemax
do i2 = 1,bgtripletmax
bgtripletangletol(i1,i2) = 10.0d0
Bharmpotbgtriplet(i1,i2) = 10.0d0
enddo
enddo
```

- 'potquartetbgflag' refers to dihedral angles between quartets of neighbor atoms

  1. 'potquartetflag' = 0: no special constraint quartet-potential active
  2. 'potquartetbgflag' = 1: constraint potential = 0 for dihedral angle within tolerance interval ('bgquartetangletol') about ideal angle, = infinity else
  3. 'potquartetbgflag' = 2: harmonic constraint potential for all dihedral angles
  4. 'potquartetbgflag' = 3: harmonic constraint potential within tolerance interval about ideal angle, = infinity else

  `potquartetbgflag = 0`
  The necessary parameters for the constraints are

  1. 'bgquartetangletol(i1,i2)' is the angle tolerance in degrees for quartet number i2 in building unit i1 (applicable for flag = 1,3).
  2. Charmpotbgquartet(i1,i2)' is the strength of the harmonic potential for quartet number i2 in building unit i1 (applicable for flag = 2,3)

```
do i1 = 1,bgtypemax
do i2 = 1,bgquartetmax
bgquartetangletol(i1,i2) = 10.0d0
Charmpotbgquartet(i1,i2) = 10.0d0
enddo
enddo
```

7 Modules and commands

We distinguish between the full program consisting of many sub-runs and the different modules one might use during an individual sub-run. The modules

are exploration procedures that range from a single energy calculation to a rather complex threshold run. They are called as commands in the command list; and at the beginning of each module, an input configuration file is created, and similarly at the end an output file. Currently the following list of modules is available:

- 'simann' = Simulated Annealing
  'mws_simann' = Multi walker separate simulated annealing
  'mwi_simann = Multi walker interacting simulated annealing

- 'thresh (Elimit,looplim)' = Threshold-Run ('Elimit' = upper energy lid; 'looplim' = number of steps)
  'mws_thresh (Elimit,looplim)' = Multi walker separate threshold run

- 'graddesc' = Gradient Descent
  'mws_graddesc' = Multi walker separate gradient descent

- 'linesearch' = Linesearch
  'mws_linesearch' = Multi walker separate linesearch

- 'quench' = Quench
  'mws_quench' = Multi walker separate quench
  'mwi_quench' = Mulit walker interacting quench

- 'multiquench' = Multiple Quench (MC/SA with T = 0 alternating with T large)
  'mws_multiquench' = Multi walker separate multiple quench
  'mwi_multiquench' = Multi walker interacting multiple quench

- 'simplex' = simplex routine (not activated?)
  'mws_simplex' = Multi walker separate simplex run

- 'powell' = powell-routine (not activated?)
  'mws_powell' = Multi walker separate powell run

- 'salocrun' = MC/SA-run with local runs shadowing
  'mws_salocrun' = Multi walker separate MC/SA-run with local runs shadowing
  'mwi_salocrun' = Multi walker interacting MC/SA-run with local runs shadowing

- path = Follow a prescribed path with local runs shadowing
  'mws_path' = Multi walker separate path run

- 'monte' = Monte Carlo run at set of fixed temperatures with search for local equilibrium
  'mws_monte' = Multi walker separate Monte Carlo run
  'mwi_monte' = Multi walker interacting Monte Carlo run

- 'mwi_partemp' = Multi walker interacting parallel tempering run
- 'calcenergy' = Calculates the energy of the current configuration
- 'findcell' = Findcell

Note that many of these subroutines are available in two or three different versions: Version one is for a single walker (e.g. called with the command 'simann'), version two is for many independent walkers that are proceedings separately, either in parallel or consecutively (e.g. called with the command mws_simann), and finally the interacting multi-walker version where all walkers interact in some fashion during the run by exchanging landscape information or via an adaptive stopping criterion (e.g. called with the command mwi_simann).

Since the subroutines that load or generate starting configurations always do so for a set of walkers (it can be only one walker in the set, of course!), the most clean way is to use just the 'mws_'-type of command also for only one walker 'nwalk = 1': As a result, you perform the run for just one walker, just as you would have for the 'simann' command.

Two important modules describe the loading in of an externally prepared (or by a previous module specially prepared) configuration, and the generation of a random configuration according to some generation parameters:

- 'loadnewcfg (confname,initdata,rsfactorflag,countinitdata1,countinitdata2)' = loading of one or many new configurations
- 'gennewcfg (confname,initflag1,initflag2,initflag3,initflag4,countinitdata1,countinitdata2)' = generating of one or many new configurations

In addition to the modules, there exist a couple of simple commands which can be used to explicitly set some parameters (pressure, random seed, moveclass, etc.) during a run:

- 'seticurr(i)' = sets the value of the single current walker to i
- 'setwalkcurr(walk1,walk2)' = sets the range of the current walker to lie between walk1 and walk2
- setseed (i) = initializes the random number generator with the seed i
- 'setnatominit (i)' = sets the number of atoms initially to i
- 'setvolfactor (r.eal)' = sets the ratio between the starting volume and the total volume occupied by the atoms when densely packed
- 'setTinit (r.eal)' = sets the initial temperature of a simulated annealing run
- 'setmoveclass (i)' = sets the moveclass explicitly to i

- 'setabinitinputflag (i)' = sets the input for ab initio calculations explicitly
- 'setfixflag (i)' = sets the value of fixflag explicitly
- 'setpress (real,integer)' = sets the pressure explicitly
- CHECK FOR OTHER COMMANDS

Finally, there is the command that tells G42+ that it is to stop the run(s),

- 'end' = end of program

Note that the name of the files that are produced by a module consist of the general 'confname' that belongs to the run, the prefix that indicates which module is being run, a counter 'countmodule' for the number of modules that have been called since the last 'loadnewcfg' or 'gennewcfg' command, and usually one or more suffixes which tell us about which part of the module we are in, e.g. which stopping point (where local minimizations take place), which minimization, etc.

7.1 Load new configuration

The subroutine 'loadnewcfg' loads one or several new configurations into the program. When this happens, G42+ first writes end configuration(s) of the previous sub-run of the program and closes the protocol. Then, the init.f subroutine is called, which initializes the new sub-run. In particular, the loadbg-subroutine loads configuration with the name given by the variable 'initdata', and starts a new protocol for a sub-run with the name given by the variable 'confname'. Note that the very first configuration can be loaded in by directly entering information in the input-file. This is usually only used for a single reference configuration that is employed to check many variables, flags and array sizes for consistency and write out the complete list of commands for the meta-program in G42+.

7.1.1 Method description

The command for loading a new configuration is
`command(..) = loadnewcfg (confname,initdata,rsfactorflag,`
`countinitdata1,countinitdata2)`
'confname' is the family name of the files that will be generated, and the number of the walker is added as a four-digit number after the confname. 'initdata' is the family name of the files that need to be read in, e.g. initdata = 'loadme'. All files that are to be loaded in have to have the name:

'loadme_????', where ???? is a four-digit number (including preceding zeroes). Thus, wherever 'hvwalk' is mentioned, this refers to a four-digit number written as four-digit character-string.

All files with values of ???? between 'countinitdata1' and 'countinitdata2' will be read in, and assigned to walkers with the numbers ????. Make sure that there are no files missing in this interval, and that 'nwalk' $\geq$ 'countinitdata2' $\geq$ 'countinitdata1' ≥ 1. Typically, the range of files will be from countinitdata1 = 0001 to countinitdata2 = nwalk.

If both 'countinitdata1' and 'countinitdata2' are set equal to zero, then only one file with the name 'initdata'(_0000) is read in and assigned to walker iwalk = countinitdata1; this case is then treated as a reference configuration for initial testing of the input of the system.

'rsfactorflag' = 1 indicates that when loading the configuration, the rs-factor is the one originally defined in input.f, and the ionic radius 'rion' is computed as 'rs*rad' for single atoms or 'rs*bgrad' for larger building units, respectively. Else ('rsfactorflag' = 0), the rs-factor is computed by dividing 'rion' from the file by the value 'rad' or 'bgrad', respectively, from the atomdatabase or bgdatabase, respectively.

7.1.2 Entries of parameters (in input.f)

If one wants to load a configuration at the beginning of the program for reference purposes, the following entries in the input-file are needed:

- `confname = 'reference'`
- `iwalk = 0`
- `initdataflag = 1`
- `natomfix = 0`
- `ntypfix = 0`

This might be needed if there are complicated settings involving background structures that would lead to contradictions if we generate the reference configuration in the usual fashion via the random structure generator.

7.2 Generate new configuration

The subroutine 'gennewcfg' generates a new configuration for a new sub-run of the program. When this happens, G42+ first writes an endconfiguration of the previous sub-run of the program and closes the protocol. Then, the init.f subroutine is called, which initializes the new sub-run. The new configuration is generated according to the value of the four parameters 'initflag1', 'initflag2', 'initflag3' and 'initflag4' (see below), and is assigned

the name given by the variable 'confname'. Of course, there are many more parameters set in the input file that control the initial configuration that is being generated. These parameters are set once and for all in the input file. Note that the very first configuration is always generated (or, alternatively, loaded in) by directly entering information in the input-file. This is usually used as a dummy configuration called 'reference', and is used to check for obvious mistakes in the metascript, databases, etc.

7.2.1 Method description

A new configuration is generated by the command
`command(..) = gennewcfg (confname,initflag1,initflag2,`
`initflag3,initflag4,countinitdata1,countinitdata2)`
Here, 'confname' is the family name of the files that will be generated and is the variable naming the new sub-run, and 'initflag1', 'initflag2', 'initflag3' and 'initflag4' are integer parameters (c.f. section on generation of random starting configurations). The number of the walker is added as (four-digit) number (in form of a character string of length 4: ????; it includes preceding zeroes) after the confname.

Thus, random configurations with values of ???? between 'countinitdata1' and 'countinitdata2' will be generated, and assigned to walkers with the numbers ????. Make sure that you want to generate walkers for the whole interval, and that 'nwalk' $\geq$ 'countinitdata2' $\geq$ 'countinitdata1' ≥ 1. Typically, the range of files will be from countinitdata1 = 0001 to countinitdata2 = nwalk.

If both 'countinitdata1' and 'countinitdata2' are set equal to zero, then only one file with the family name 'confname' is generated and assigned to walker iwalk = countinitdata1 (= 0000). Again, this is used as a reference configuration for testing purposes.

7.2.2 Entries of parameters (in input.f)

If one wants to generate a configuration at the beginning of the program, the following entries in the input-file are needed (of course, one can choose different values for 'initflag1', etc.):

- `confname = reference`
- `iwalk = 0`
- `initdataflag = 0`
- `natomfix = 0`
- `ntypfix = 0`

- `initflag1 = 1`
- `initflag2 = 1`
- `initflag3 = 0`
- `initflag4 = 0`

7.2.3 Parameters directly or indirectly involved in generation of new configuration

Several parameters are set in input.f outside of the actual 'gennewcfg' command that influence the randomly generated configuration. Of course, there are parameters like the number of atoms and building units of various types, but there are some that are only specific for the generation of the cell and the atom positions inside the cell. These are given here:

- Prescription of initial cell parameters

  1. 'volinitflag' indicates whether initial cell parameters should be prescribed. For different moveclasses i1, different choices of 'volinitflag' are possible.
     `volinitflag(i1) = 1`: Simulation cell must have parameters given by 'ainit'
     `volinitflag(i1) = 2`: The volume per atom is prescribed 'volperatom', and G42+ generates the cell accordingly.
     `volinitflag(i1) = 0`: The cell is not generated with fixed cell parameters.
     If we continue our run by loading in a new configuration, then the cell parameters are always taken from this configuration. If you use 'backgroundflag' = 1 (add a fixed background structure), then all 'volinitflag' must be set equal to 1 and all 'ainit' must be the same cell (for all walkers) which must be identical to the cell of the background structure.
  2. 'ainit(i1,i2,i3,i4)' is the cell parameter 'a(i2,i3)' for walker i1 and moveclass i4, which is prescribed from the outset if 'volinitflag' = 1. Here 'a(i2,i3)' is the i2'th Cartesian component of the i3'th cell vector. Note that the matrix 'a(...,...)' is the transpose of the matrix 'ag(...,...)' used in the configuration file!
  3. 'volinit' should be given and be equal to the volume computed from 'ainit' ($\pm$ 1.0). This is just a check. If the two values 'volinit' and the volume from 'ainit' do not agree, a warning is written, and the values of 'ainit' are assumed to be the correct ones.
  4. 'volperatom' is the volume per atom for an orthorhombic or cubic cell with volume natominit*(volume/atom)

- 'placetry' is the number of attempts to place one atom or building unit without overlap. `placetry = 5000`

- 'volfactor' determines for random cell generations, by what factor the starting cell volume should be larger than the volume *vol* of all the atoms together, $vol = \sum_j V_j$, where the atoms are considered spherical, $V_j = (4\pi/3) rmax(atype(j), ch(j))^3$; based on this, the length of the cell vectors 'a(i,j)' are computed. Note that the cases of only atoms (e.g. 'volfactor' = 3) and ion cores plus electrons are different (e.g. 'volfactor' = 1).
  Furthermore, note that here the atom volume refers to the largest (!) possible ionic radius among those available for the atoms of type 'atype' that participate, and not to the radius of the initial atoms or ions; especially for feasible large values of the ionic charge, this can be quite different from the neutral atom radius (c.f. atomdatabase). Thus, sometimes, even rather small values of 'volfactor' such as 1.0 can be sensible, if one wants to start with a density close to the final solid for neutral atoms instead of the more gaseous densities usually used for long simulations.
  Also, one would usually pick 'volmaxfact' such that the maximal cell size is larger than 'vol'*'volfactor'.
  Keep in mind that 'volfactor' is not active if 'volinitflag' = 1 or 2, i.e. if the initial cell is prescribed at the beginning.
  Finally, note that if we deal with layers of atoms, then we can fix the z-direction c (= a(3,3)), and use 'volfactor' to compute the cell-parameters in the xy-direction (see below).
  `volfactor = 1.0d0`
  We also have the option to enter a value for 'volfactor' from an external file without having to recompile the code, using the flag 'readinflag4' ('readinflag4' = 0: no external data):
  ```
  if ((readinflag4.eq.1).and.(readinflag.eq.1)) then
  volfactor = hvdat4
  endif
  ```

- Next we come to tolerance in the random generation of configurations (once the cell has been generated or taken from 'ainit':

  1. 'tol3' is the minimal distance allowed during the initialization between building units. If these correspond to individual atoms, then 'tol3' should have the same value as 'tol4'. Note that 'tol3' is not automatically compared with 'mindist'; if you use a 'mindist'-potential, then one should go with 'tolatomflag' =1 and compare with the atoms themselves, unless 'tol3' is already larger than all the 'mindist' involved. Since 'tol3' refers to distances between the reference points of whole building units, then for large building

units, use larger values of 'tol3' to avoid big overlaps once the atoms are put in.
`tol3 = 0.1d0`

2. 'tol4' controls the overlap of atoms directly. For very large and convoluted building units, it might be necessary to base the decision whether a randomly placed building unit is compatible with the rest of the structure on the overlap of the atoms. This is measured by the value of 'tol4'. In addition, we set 'tolatomflag' = 1, to make sure that not only the centers of the building units are far enough from one another, but that also all atoms do not overlap too much. In that case, we also check whether there is overlap between the atoms of a building unit with atoms in a neighbor cell since the building units can reach into neighboring cells; thus here we do not need to set 'rimflag' = 1 (see below).
   Note that if one selects 'pottype' = 12 (and 'potflag' = 1, of course), then the program automatically uses the larger value of 'tol4' and the predefined values in 'mindist' as tolerances when generating a random starting configuration. Further note that the program does not prevent you from loading in explicitly a configuration that violates 'mindist'! Also, one should avoid a building unit to have atoms that lie beyond perfect slabs of some kind ('slabmirrorflag' or 'slabvdWflag' > 0). Thus, one should set 'tolatomflag' = 1 for such situations to be on the safe side.
   Note furthermore that we would get into trouble, if we intertwine two large molecules and at the same time perform calculations that only refer to atom positions such as ab initio energy calculations because then the molecules will be reorganized and atoms from one molecule will be assigned to another one!
   `tol4 = 0.1d0`

3. The distance of the atoms away from the perfact slab is given by 'tol5' in Angstrom.
   `tol5 = 1.0d0`

4. 'tolatomflag' indicates whether it is checked that the atoms of two building units are sufficiently far away from each other when we generate the initial configuration. 'tolatomflag' = 0: no check; 'tolatomflag = 1': make check.
   `tolatomflag = 0`

5. 'rimflag' checks whether a rim at the border of the unit cell can be ('rimflag' = 0) or cannot be occupied by a randomly placed atom ('rimflag' = 1) for an (orthorhombic) simulation cell. The size of the rim is given by ('tol3')/2. Usually, rimflag = 1 makes sense, but when using certain kinds of slabs, occupation of only planes, or fixed background structures, it might make sense to set

'rimflag' = 0. Also, if we use 'tolatomflag' = 1, then the rim is already checked, and we can set 'rimflag' = 0, if we want to.
Note that if you use a prescribed initial cell that is not orthorhombic, then 'rimflag' = 1 is not very useful, since it can be tricky to compute the right kind of rim size. In that case, use 'tolatomflag' = 1, in order to take overlaps at the rim of the cell into account.
`rimflag = 0`

- Atoms in layers as starting configurations

  1. 'zinitflag' allows us to place atoms into layers parallel to the xy-plane: If we want to place the atoms randomly in 'n_zlayers' layers with the same z-value, we set 'zinitflag' = 1, 'zinit(i1,i2)' = desired value of layer i1 for walker i2.
     This is usually used in combination with `move2dflag = 1`, i.e. if the atoms are only allowed to move in the x-y-plane, and if only the cell parameters 'a(1,1)', 'a(1,2)', 'a(2,2)' and 'a(2,1)' are allowed to be changed. For this we would preferably use only monoclinic cells that have 'a(2,3)' = 'a(1,3)' = 'a(3,2)' = 'a(3,1)' = 0.
  2. Starting cells can either be prescribed at the beginning ('volinitflag' = 1) in toto, or we can prescribe only the length in z-direction by giving 'L_z', and the two remaining cell vectors are generated using 'volfactor' restricted to 'n_zlayers' planes, i.e., with respect to the maximal area occupied by the atoms j: $A = \sum_j A_j$; $A_j = \pi * (rmax(j)^2)$. In that case, 'volinitflag' must be equal to zero, of course.
     Filling more than one layer can be tricky, since for 'move2dflag' = 1 no moves are allowed to move atoms between the layers, and thus the number of atoms within each layer are fixed. (Note that since we can exchange atoms between layers, the types can be changed; if one wants to also change the number of atoms, one needs to provide lots of vacancies that can be exchanged for an atoms between layers.) Making it possible for atoms to jump between layers without the help of 'dummy vacancies' is currently only a future option, so one should use now only 'n_zlayers' = 1. If one wants to work with several layers with definite numbers of atoms per layer, where atoms can only move within the assigned layer, one would prepare such an initial configuration file by hand.
  3. If we use random assignment of atoms positions, then we randomly distribute all atoms over all layers for 'zinitflag' = 1. For 'zinitflag' = 3, 'numatoms_zinit(i1,i3)' gives the number of atoms of type i3 that are prescribed as initial number within layer i1. If

we set 'zinitflag' = 2, then the total number of atoms per layer is set equal to 'numatoms_zinit(i1,0,i2)'; if we set 'zinitflag' = 3, then the number of atoms of type i3 per layer is equal to 'numatoms_zinit(i1,i3,i2)' for walker i2. Make sure that the number of atoms per type defined for the whole system in 'initntype' correctly takes the distribution of atoms over the layers into account! (Note that we have not distributed atoms according to their charge, only acccording to type.)
Note that if you combine 'zinitflag' = 1 with prescribed slabs or a prescribed background structure, you need to make sure that the values of 'zinit' are compatible with the restrictions of atoms on slabs. In particular, keep in mind that randomly placed (!) atoms are supposed to be at z > 0.6 (or 0.6 < z < 0.9 for two slabs). (Possibly one needs to change the value of 0.6 explicitly in the code for some situations - DEFINE AS PARAMETER IN INPUT.F.) Currently, the choice of 'zinit' overrides this requirement as long as 'zinit' is larger than 0.5 (and smaller than 1.0 for two slabs)! Furthermore, for this kind of situation, the rim of the unit cell is off limits (i.e. as if 'rimflag' were set to zero).
Note: As mentioned above, if you want to perform a quasi-2d-optimisation, where the atoms are only to be moved within planes, but you do not have a background structure to worry about, and the cell parameters in the xy-plane can change and should be generated at random at the beginning, then use 'volinitflag' = 0 and prescribe the correct value for the c-axis by 'L_z'. If we do not use 'zinitflag' > 0, but still want to have L_z > 0, then we must set 'Lzflag' = 1.

```
zinitflag = 0
n_zlayers = 1
do i2 = 0,nwalkmax
zinit(1,i2) = 0.5d0
numatoms_zinit(1,0,i2) = 12
numatoms_zinit(1,1,i2) = 6
numatoms_zinit(1,2,i2) = 6
L_z(i2) = 1.6d1
enddo
Lzflag = 0
```

- Flags for type, charge and position allocation of atoms/ units

  1. 'initflag1' determines whether the atoms are given their types from a distribution or according to pre-fixed numbers: 0 = chosen randomly according to a distribution described by 'tflag' etc. 1 = 'initntype(i)' atoms of type i allocated one after the other. 2

= 'initntype(i)' atoms allocated in a regular fashion. In this case, the type of the biggest number must be listed first.
Currently, only 'initflag1' = 1 is active, since we do not work with variable composition.
`initflag1 = 1`

2. 'initflag2' determines whether the charges are allocated according to a distribution or according to pre-fixed numbers. Do not use pre-fixed numbers in 'initflag2' together with random allocation in 'initflag1'. 0 = allocate randomly according to distribution determined by 'ctflag' etc. > 0 = allocate according to pre-fixed numbers 'initnctype(i,j)' of amount of charge j for type i. Currently, only 'initflag2' = 1 is active.
`initflag2 = 1`

3. 'initflag3' determines whether the original positions of the atoms are supposed to be randomly or equally spaced; 0 = random positions; 1 = random positions but distances between atoms > sum of ionic radii * 'fact1'. 2 = equally spaced. Note that if we use 'initflag3' = 1 for building units of size larger 1, then we need to use 'tolatomflag' = 1. Furthermore, remember to set 'fact1' according to the amount of overlap that is to be allowed. In this context, also check, which potential is being used, to make sure that there are no contradictions. The setting 2 is currently not active.
`initflag3 = 0`
fact1' gives the fraction of the sum of the (ionic) radii of the atoms that are being inserted, which is allowed for 'initflag3' = 1 as minimium separation between atoms during the initialization stage.
`fact1 = 1.0d0`

4. If we want to not only place the building units at random positions, but also rotate them about an arbitrary angle, then set 'rotateinitflag' to 1 (0 = no rotation)
`rotateinitflag = 0`

5. 'initflag4' determines, whether there shall be a randomization of positions before the actual simulated annealing run occurs. The length of the run is determined by the value of 'initflag4': 0 = no randomization run, n = Each atom makes n moves in a random direction (or is exchanged with another atom etc.). Use 'stepsmax0' (< 'stepsmax'), if we use 'flagT' = 1. Currently, 'stepsmax0' > 0 is not active!
`stepsmax0 = 0`
`initflag4 = stepsmax0`

7.3 Calculate single energy

The subroutine 'calcenergy' makes the program perform a single energy calculation for the current walker.

7.3.1 Method description

The command call for 'calcenergy' is:
`command(..) = calcenergy`
No parameters are set when using this command, i.e., all energy calculations are performed using the potential currently being active according to the input to G42+. The prefix indicating that the output files have been generated during a single energy calculation is 'onece'.

7.3.2 Entries of parameters (in input.f)

No parameter entries at the current time.

7.4 Simulated annealing

The simulated annealing module is essentially a random walk for a decreasing sequence of temperatures. The attempted moves are selected from a moveclass, and they are accepted according to the Metropolis criterion. In addition, further criteria can be given, such as minimum/maximum energy difference thresholds, absolute values for minimum/maximum energies, etc. The size of some of the moves can be reduced with decreasing temperature, in order to allow for a more efficient refinement of the minima (see section moveclass for more details). The temperature in the acceptance criterion is reduced either by following a prescribed schedule or adapted according to the success of the run. Similarly, the simulated annealing module finishes either after some prescribed length or in an adaptive fashion.

As for all 'mws_'-type runs, each of the (nwalk) walkers performs its own individual simulated annealing run. They can either be performed consecutively (when one runs the G42+ part of the calculations on only one processor) or in parallel distributed over several processors (if the G42+ part is on several processors but the external codes - if employed - only run on one processor each).

For the 'mwi_'-type runs, all walkers must run in parallel, of course. The following information exchange can take place:

- Cross-over moves in the moveclass, where a pair of two walkers generates two new candidate configurations by taking part (half) of the configuration from one walker and put it together with the complementary part (half) from the other. For more details, c.f. the description of the multi-walker moveclass 'moveclassmulti'.

- Penalty terms (= increase in energy) when the configuration of one walker becomes too similar to the configuration of another walker. This is meant to keep up diversity during the exploration such that each basin is only studied by one walker, preferably.
- Multi-walker criterion for the decision to lower the temperature during the simulated annealing run and/ or to stop the run.

Note that options in the penalty function which automatically keep track of the results of previous searches and save these minimum configurations as static penalty configurations (if you get too close to one of them, your energy increases rapidly) for the next run will probably crash if they are run for more than one walker at a time ????????CHECK????????

7.4.1 Method description

The command call for simulated annealing is:
`command(..) = simann`, `command(..) = mws_simann`, `command(..) = mwi_simann`.
No parameters are set when using this command, i.e., all simulated annealing runs will follow the set of parameters defined in the input to G42+ (note that most of these can be different for different walkers). The prefix indicating that the output files have been generated during a simulated annealing run is 'onesa', 'mwssa' or 'mwisa', respectively.

During the run, it is possible to periodically stop the walker at intermediary configurations (holding points) along the trajectory and perform one or more local runs (typically local minimizations). The number of times to stop along simulated annealing for walker i1 is given by 'saquenchloop(i1)' (must be smaller than 'saquenchloopmax' set in 'varcom.inc'):

```
do i1 = 0,nwalkmax
saquenchloop(i1) = 10
enddo
```

In principle, each walker can have a different number of holding points. 'msa2max' is the number of times we quench from a holding point, e.g. `msa2max = 2`.

'satypeflag' indicates, which kind of local run-algorithm is to be used in the local-runs that starts from the holding points: INTRODUCE VIA LOCALRUN INSTEAD

- `satypeflag = 0`: standard stochastic quench
- `satypeflag = 1`: gradient steepest descent with built-in potential
- `satypeflag = 2`: simplex-method (not yet activated)
- `satypeflag = 3`: line search
- `satypeflag = 4`: powell-method

- `satypeflag = 5`: no stepwise minimization

As discussed below in the local-run module, for such local explorations we employ a local moveclass, the number of which is given by the value of the global variable 'mclflag_local' (Remember that we can in that case use e.g. another energy function, too!). However, we might want to perform this in-between local mimization with a different moveclass than this standard local moveclass, or perhaps with the same moves as prescribed for 'mclflag_local' but a different energy function such as an ab initio energy. In that case, we can pick our own special moveclass by setting the variable 'samclflag_local' to the number of the moveclass. If 'samclflag_local' = 0, then we take the usual 'mclflag_local' moveclass, which is the standard case, `samclflag_local = 0`.

If we want to perform a quench run, gradient steepest descent with built-in potential, or line search with built-in potential, for an intermediary configuration over a certain fixed number of steps, we set 'saquenchnum' = number of steps of these runs. If no such run is intended, set `saquenchnum = 0`, or `satypeflag = 5`. This is useful, if we want to follow up the local minimization with the built-in moveclass (with the specific choice of energy function, fixed atoms, etc. that can be prescribed via the choice of moveclass) by a full local gradient based minimization using either the built-in potential or an external code: If we want to perform a gradient descent after the quench for each local best configuration, set `sagraddescflag = 1`. If one only wants a gradient descent, use `satypeflag = 5` and `sagraddescflag = 1`. If `sagraddescflag = 0`, then no such gradient minimization takes place after the actual local run.

7.4.2 Entries for parameters (in input.f)

The main issue is the control of the temperature during the simulated annealing: choice of cooling schedule, length of run, criterion of switching between temperatures. In principle, nearly all of these parameters can be chosen to be different for different walkers.

- 'Tinf' defines the temperature value which equals to infinity (real number). This acts as a reference value in various subroutines. In particular, the ratio of log10('Tinit')/log10('Tinf') controls the initial value of the step size for atom positions and cell parameters, 'dT' and 'dcellT', respectively, in the tupdate subroutine. (for other values of T, the ratio is log10(T)/log10('Tinf'), of course). Typically, we choose 'Tinf' = 10**5 * 'Tinit', e.g. for various quench runs.
  `Tinf = 1.0d5`

- 'flagT' determines which temperature schedule is to be used.

  1. `flagT = 1` : T = 'Tinit' * ('decrease1')**nT;

2. `flagT = 2` : T = 'Tinit' / (nT * 'decrease2' + 1);
3. `flagT = 3` : Huang adaptive schedule; For details, c.f. tupdate.f or input.f
4. `flagT = 4` : Schedule similar to Huang, but with T**2 instead of T in the (differential) equation; For details, c.f. tupdate.f or input.f
5. `flagT = 5` : T = T - (T * 'decrease5') (own schedule);
6. `flagT = 6` : linear schedule: T = T - 'decrease6' (best for comparison with quench experiments);
7. `flagT = 7` : T = const. (corresponds to T = 'Tinit' and thus produces a Monte Carlo run with constant temperature: use only together with 'abbrflag' = 0, i.e. with a fixed total number of steps !!)

- Some values for the decrease parameters might be

  1. `decrease1 = 0.995d0`
  2. `decrease2 = 2.0d0`
  3. `decrease3 = 0.005d0`
  4. `decrease4 = 0.005d0`
  5. `decrease5 = 0.01d0`
  6. `decrease6 = 0.0001d0`

  Note that 'decrease1' (real number) < 1, 'decrease2' (real number) > 0, in order to make sense.

- 'Tinit' is the initial temperature (real number).
  `Tinit = 1.00000d0`

- Parameters for the adaptive choice of 'Tinit':
  The subroutine 'init5' performs simulated annealing steps at a temperature T (starting with 'Ttest'), and adjusts the temperature depending on whether the fraction of accepted moves lies within the interval 'accmin' and 'accmax'.
  `init5flag = 0`: no adjusting;
  `init5flag = 1`: use initialization routine.
  'init5steps' is the number of steps at T, 'init5max' = number of attempts to optimize 'Tinit'. 'init5fact' > 1 is the factor to increase the test temperature (1 / 'init5fact' decreases). Typical values may be:

  1. `init5flag = 0`
  2. `Ttest = 1.0d-3`

3. `accmin = 0.95d0`
4. `accmax = 0.99d0`
5. `init5fact = 1.5d0`
6. `init5steps = 1000`
7. `init5max = 20`

- 'Tlimit' is the absolute lower limit to which the temperature can go. `Tlimit = 0.00001d0`
- The next flags serve for the precise setting of the stopping criterion of the simulated annealing. 'abbrflag' gives the general class of stopping criterion:

  1. `abbrflag = 0`: Stop after 'nTmax' temperature steps;
  2. `abbrflag = 1`: Stop if the average energy after 'numTchange' temperature changes has not changed by more than a factor 'dchangeEav';
  3. `abbrflag = 2`: Stop if the average of the best energies so far 'Ebest' after 'numTchange' temperature changes has not changed by more than a factor 'dchangeEbest';
  4. `abbrflag = 3`: Stop if 'Ebest' after 'numBchange' temperature changes has not changed by more than a factor 'chEbest';
  5. `abbrflag = 4`: Stop when temperature 'Tlimit' has been reached.

  For options '2' and '3', the computation of the average for the check of the stopping criterion only begins after 'droptime' loops, unless 'droptime' $>$ 5000 or 'Ebest' $<$ 10. Note that 'numTchange' must not exceed Ebstmax (set in 'varcom.inc'), and similarly 'numBchange' must not be bigger than EBmax (set in 'varcom.inc').
  During option '3', the check of the stopping criterion is always activated for T $<$ 0.0001 K.
  Note that if one only wants to perform a single energy calculation, one can use 'abbrflag' = 0 and 'nTmax' = 0 (this is an alternative to using the command 'calcenergy').

  Next, we list the variables needed for the stopping criterion:

  1. `nTmax = 1000`: Number of temperature changes, needed for 'abbrflag' = 0
  2. `nTmaxlimit = 10000000`: Maximum number of temperature changes to avoid infinite loops
  3. `dchangeEav = 0.00001d0`: Change in the average energy, needed for 'abbrflag' = 1

4. `dchangeEbest = 0.0001d0`: Change in the best energy observed, needed for 'abbrflag' = 2
5. Parameters for 'abbrflag' = 3:
   `chEbest = 0.0000000001d0`
   `numBchange = 50`
6. Parameters needed for equilibrium detection:
   'equildetect' indicates whether G42+ should run for so long at one temperature, until equilibrium at that temperature has been reached (= 1, detector active), or if the temperature should always change after 'stepsmax' steps (= 0, detector inactive, 'stepsmax' must be smaller than 'stepsmainmax').
   Equilibrium is suppose to have been reached, if the average energy after at least 'numEchange'+1 steps becomes bigger that the average energy had been 'numEchange' steps ago. Note that 'numEchange' must not exceed Eeqmax (set in 'varcom.inc')
7. `stepsmax = 100`: Number of loops between temperature changes if 'equildetect' = 0
8. `stepsmaxlimit`: maximum number of such loops needed for multi-walker implementations, to ensure smooth stopping. It is usually chosen to be equal to 'stepsmax' plus one.
9. `numEchange = 100`: Parameter for 'equildetect' = 1

- Next, we deal with checks on the energy range where the exploration is taking place.

1. we can check, whether a given minimum energy has been reached, and stop the simulated annealing:
   `Eminimumlimitflag = 1`: check whether the walker has reached 'Eminimumlimit'.
   This is useful, if one knows, where the energies of low-lying minima in the system are, and thus when it is time to stop the simulated annealing and start e.g. a quench. Note that this is different from rejecting moves that reach unphysical energies, or from forcing the walker to stay in an energy band, 'EMCbottom' - 'EMCtop'.
2. 'cutoff1' is the value for the ratio deltaE/T above which the move is always rejected. Similarly, 'cutoff2' is the value of deltaE/T below which the move is always accepted. 'cutoff2' = 0.0 corresponds to regular simulated annealing. 'cutoff2' = 'cutoff1' gives the 'threshold accepting algorithm'. Typical values of these variables might be:

```
cutoff1 = 10.0d0
cutoff2 = 0.0d0
```

3. 'Einf' is an infinitely large energy, which is assigned to configurations that are unphysical and should be forbidden, such as e.g. too high ionization energies, or atoms getting too close, without having to compute all the other terms in the energy function of a configuration. A typical value would be
   `Einf = 1.0d+13`
4. 'Eflag' determines whether the program shall use the full 'energy'- or the so-called 'deltaE'-subroutine in order to calculate the energy difference (useful when only a few atoms are moved such as for a simple atom exchange or a single atom move).
   `Eflag = 0`: use energy routine
   `Eflag = 1`: use deltaE routine
   The deltaE-subroutine works by computing only the terms in the energy that belong to those atoms that are being moved, both before and after the move, and taking this difference. It assumes that the energy function is written as a simple sum of two-body and possibly multi-body potentials (such as the Gupta-potential).
   The option for using the deltaE-routine is at the moment not active for moves involving changes of the cell (all atoms are affected!), variations of stoichiometry, simulations with free parameters (non-trivial space groups), and runs along a prescribed path. Furthermore, we cannot use the deltaE-routine for calculations with external codes, of course. Thus, before using the deltaE-routine, check whether it is available for kind of calculations planned.
5. If E is outside of the allowed interval between 'EMCtop' and 'EMCbottom', the move gets rejected during simulated annealing, Monte Carlo or threshold runs. In order to allow decrease from some random position with energy outside the energy band, this criterion does not apply, if Ecurrent (the previous energy), is also above/below the energy lid. For standard simulated annealing, we only use these values to eliminate crazy energies:
   ```
   EMCtop = 1.0d13
   EMCbottom = -1.0d13
   ```

7.5 Stochastic quench

The stochastic quench is a random walk with T = 0. In order to allow for more efficient search using smaller steps, a pseudo-temperature is defined according to which the step-size is adjusted just as the moveclass in simulated annealing.

7.5.1 Method description

The command call for the quench is: command(..) = 'quench'. No parameters are set when using this command, i.e., all regular quench runs will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a quench run is 'onequ', 'mwsqu' and 'mwiqu', depending on whether we are running a quench for a single walker, a set of independent walkers, or a set of interacting walkers, respectively.

7.5.2 Entries of parameters (in input.f)

- The parameter 'numsteps' denotes the number of steps for a quench or multi-walker quench. If 'numsteps' = 0, 'lpquench' is used as stopping criterion. In order to avoid endless loops, 'numstepslimit' gives the absolute limit in the number of allowed quench steps. Note that 'numsteps' can be read in from an external file to avoid recompilation if 'readinflag5' = 1.

```
numsteps = 1000
readinflag5 = 0
if ((readinflag5.eq.1).and.(readinflag.eq.1)) then
numsteps = hvdat5
endif
numstepslimit = 1000000
```

- 'wrstep' gives the number of steps between output in multiquench and quench, and also in prescribed path.

```
wrstep = 100
```

- 'lpquench' denotes the number of steps, during which one waits for an improvement in energy. If during this interval nothing has happened - defined by the energy not improving by more than 'qulimit' -, the quench (both a single-walker, and, if all walkers show no change, a multi-walker quench) is stopped. 'lpquench' should not exceed 'EBmax' which is set in 'varcom.inc'. Note that 'lpquench' and 'qulimit' are the values also used when running a quench inside 'localrun' and 'palocalrun' using this adaptive criterion for stopping.

```
lpquench = 20
qulimit = 0.001d0
```

- 'quTdecrease' indicates, by which factor the control temperature is decreased after 'lengthquloop' quenchsteps, T(n+1) = T(n) * 'quTdecrease'. This temperature is used to adjust the step lengths in the moveclass towards the end of the quench to smaller values from initially rather large ones. 'quTdecrease' = 1 implies T = const. (= 'Tquinit'). 'Tquinit' is the starting temperature of each quench run.

'Tlocalqu0' is the temperature of a local run that may be called from the mwi-quench module. Note that all these values can be chosen differently for different walkers; however, 'numsteps' and 'numstepslimit' must be the same, in order to allow interacting runs to proceed.

```
do i1 = 0,nwalkmax
quTdecrease(i1) = 1.0d0
Tquinit(i1) = 1.0d0
Tlocalqu0(i1) = 1.0d0
enddo
lengthquloop = 100
```

7.6 Gradient descent

The gradient descent implements a steepest descent in the direction of the gradient of the configuration. Note that both atom-positions and cell parameters are included in the gradient for periodic systems. If you are using the built-in gradient, then, after the gradient has stopped, we perform a local exploration along all coordinate directions, in order to check, whether we are in a real minimum or on a saddle point. Interacting multi-walker runs do not make sense. Note that if we are using external codes for the energy computation, then the gradient descent minimization needs to be run via these codes since we cannot extract gradients from the external programs. However, not all these external codes contain appropriate gradient and minimization routines.

7.6.1 Method description

The command call for the gradient is: command(..) = 'graddesc' or command(..) = 'mws_graddesc' for several parallel non-interacting walkers, respectively. No parameters are set when using this command, i.e., all gradients will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a gradient descent run is 'onegd' or 'mwsgd', respectively.

7.6.2 Entries of parameters (in input.f)

- 'lpgradient' denotes the maximal number of steps allowed for a full gradient run.
  `lpgradient = 1000`
- 'lpoutputgrad' gives the number of steps between write commands in the gradient routine.
  `lpoutputgrad = 1`
- 'dstep0' denotes the basic stepsize of the gradient routine.
  `dstep0 = 0.01d0`

- If the energies of two consecutive configurations in the gradient descent differ by less than 'gdlimit', then the gradient descent will be stopped.
  `gdlimit = 1.0d-6`
- checkstep is the size of the step away from a possible minimum configuration in the gradient methods, in order to find out, whether we are really dealing with a local minimum.
  `checkstep = 0.001d0`
- 'normlimit' is the norm of a gradient, below which the check routine for a local minimum is activated.
  `normlimit = 0.001d0`

7.7 Line search

Instead of re-evaluating the gradient after each step, we proceed along the direction of the gradient until a minimum along this line in configuration space has been reached. Then the gradient is again computed. Interacting multi-walker runs do not make sense.

7.7.1 Method description

The command call for the linesearch is: command(..) = 'linesearch' or command(..) = 'mws_linesearch' for several non-interacting walkers, respectively. Again, interacting walkers do not make sense. No parameters are set when using this command, i.e., all linesearches will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a line search run is 'onels' or 'mwsls'.

7.7.2 Entries of parameters (in input.f)

- 'dstepls0' is the basic step of a line search gradient.
  `dstepls0 = 0.01d0`
- 'lpoutputls' is the number of steps between write commands in the linesearch routine.
  `lpoutputls = 1`
- 'lsloopmax' = maximum total number of steps within a line search ('lsloopmax' $\geq$ 'lplinesearch' * 'climit').
  `lsloopmax = 5000`
- 'lplinesearch' is the number of line searches in the line search routine.
  `lplinesearch = 10`
- 'climit' is the maximal number of steps between two gradient evaluations (for a line search).
  `climit = 50`

7.8 POWELL

See literature (numerical recipes) for description of POWELL minimization routine. Interacting multi-walker runs do not make sense. (CURRENTLY NOT ACTIVE)

7.8.1 Method description

The command call for the POWELL is: command(..) = 'powell' or command(..) = 'mws_powell' for non-interacting multi-walker runs, respectively. No parameters are set when using this command, i.e., all POWELL searches will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a POWELL run is 'onepo' or 'mwspo'.

7.8.2 Entries of parameters (in input.f)

Note that many of the variables in the POWELL-routine are denoted 'detsea_'... (short for 'deterministic search' that is based only on energy evaluations).

- Relative uncertainty of minimum.
  `detsea_ftol = 1.0d-14`
- Starting values for step-size in powell for axes and position coordinates.
  `detstepa = 1.0d-1`
  `detstepx = 1.0d-2`
- maximum number of 'ATOM_EXCHANGE_RELAX'-calls.
  `detsea_naermax = 3`
- maximum number of cycles of searches along all directions within POWELL.
  `detsea_ncipmax = 20`
- maximum number of POWELL calls.
  `detsea_npcmax = 60`
- Parameters in the subroutine 'MNBREAK':
  `gold = 1.618034d0`
  `glimit = 100.d0`
  `itermax = 30`
  `tiny = 1.0d-20`
- Parameters in the function 'BRENT':
  `itbrmax = 100`
  `cgold = 0.381966d0`
  `zeps = 1.0d-4`

- Parameters in function 'LINMIN':
  `TOLlm = 1.0d-3`

7.9 Simplex

See literature (numerical recipes) for description. This routine is not yet activated.

7.9.1 Method description

The command call for the simplex routine is: command(..) = 'simplex'. No parameters are set when using this command, i.e., all simplex searches will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a simplex run is 'onesx'.

7.9.2 Entries of parameters (in input.f)

No entries, since not activated.

7.10 Monte Carlo

The Monte Carlo routine performs constant temperature random walks using the Metropolis criterion. The temperatures are drawn from a list or generated by some rule (see below). Along these runs, we use observation time windows, in order to find candidates for locally ergodic regions at a given temperature, i.e. we check whether we are in local equilibrium. As indicator observables serve the potential energy/enthalpy and the pair correlation function (not activated).

We average over the time intervals, and compare the difference in the observables from one interval to the next with the fluctuations within the two intervals under consideration. Once we observe the system to get into local equilibrium, we register the configuration with the lowest energy where this happens, for future analysis. In addition, we register for each temperature the configuration with the lowest energy for the whole run.

Periodically, we check along the run, which kind of local minima are residing 'below' the trajectory.

We can restrict the run to be between two energy levels, like in simulated annealing ('EMCbottom' and 'EMCtop').

In the case of multiple interacting walkers, the only interaction is the penalty for getting too close.

7.10.1 Method description

The command call for the Monte Carlo run is: command(..) = 'monte', or command(..) = 'mws_monte' or command(..) = 'mwi_monte', respectively.

No parameters are set when using this command, i.e., all Monte Carlo runs will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a Monte Carlo run is 'onemc', 'mwsmc' or 'mwimc'.

7.10.2 Entries of parameters (in input.f)

- 'mcintmax' is the number of intervals employed for a given temperature. It should be smaller than 10000. The length of each interval is 'mcintlength'. If we only want to start with the analysis of intervals after 'mcintdelay' intervals have elapsed, set 'mcintdelay' to that number. Else, set 'mcintdelay' equal 0. Never put in a negative number for 'mcintdelay'.

```
do i20 = 0,nwalkmax
mcintmax(i20) = 100
mcintlength(i20) = 10000
mcintdelay(i20) = 50
enddo
```

- 'mcflagT' controls the way we pick the temperatures for the sequence of Monte Carlo runs.

  1. 'mcflagT' = 0: Read from an array
  2. 'mcflagT' = 1: T = mcTinit*(mcdecr)**mcloop
  3. 'mcflagT' = 2: T = mcTinit/(mcloop*mcdecr+1.0d0)
  4. 'mcflagT' = 3: T = mcTinit - mcloop*mcdecr

  ```
  mcflagT = 0
  ```

  The number of temperatures employed is given by a) 'mcloopmax', b) 'mcTlimit', c) 'mcnumTarray'. Unless we pick the temperatures from a pre-defined array called 'mcTarray', we stop if either 'mcloopmax' or 'mcTlimit' has been reached. If we pick from an array with 'mcnumTarray' entries, we stop once these have been run through. 'mcloopmax' must be smaller than 1000, 'mcnumTarray' smaller or equal 'tempmax' (given in 'varcom.inc').

  We start counting the runs for different temperatures with the internal counter 'mcloop' set to 0, unless we read from the temperature list (then we start with 'mcloop' equal 1). The starting temperature is 'mcTinit'.

```
do i20 = 0,nwalkmax
mcloopmax(i20) = 10
mcTlimit(i20) = 0.1d0
mcTinit(i20) = 1.0d0
```

```
mcnumTarray(i20) = 10
enddo
```

Note that we can pick different temperature programs for different walkers, thus covering many more temperatures than when giving each walker the same list of temperatures. On the other hand, we might want to establish a certain number of samples for the same run conditions but with different random number sequences, of course.

```
do i20 = 0,nwalkmax
mcTarray(i20,1) = 1.0d0
mcTarray(i20,2) = 0.8d0
...
mcTarray(i20,10) = 0.01d0
mcdecr(i20) = 0.05d0
enddo
```

- 'mcmoveTflag' is the analogous flag to 'moveTflag' in the other types of random walk based runs. It tells us, whether to adjust the step-size according to some rule ('mcmoveTflag' = 1) or keep the values fixed ('mcmoveTflag' = 0).
  `mcmoveTflag = 0`

- 'mccurnum' is analogous to 'savecurnum'. Every 'mccurnum' steps we write out the current configuration. 'mccurnum' = 0 means that nothing is written (recall that these configurations are being overwritten every time, so if the machine uses a large memory buffer before writing out a new configuration file, the old one in the buffer will have been overwritten already.
  `mccurnum = 0`

- 'mcwatchenergy' indicates whether the average energy and standard deviation sigma should be printed for each interval (=1) or note (= 0).
  `mcwatchenergy = 1`

- If we want to perform a quench run for every local best configuration that appears to be in equilibrium, we set 'mcquenchnum' = number of steps in the quench run. If no quench run is intended, set 'mcquenchnum' = 0.
  `mcquenchnum = 10000`

- If we want to perform a gradient descent after the quench for each local best configuration, set 'mcgraddescflag' = 1.
  `mcgraddescflag = 1`

7.11 Parallel Tempering

Parallel Tempering performs multi-walker runs with each walker at a different temperature and pressure. As usual, all walkers are random walkers. Periodically, they switch temperatures and pressures, either automatically (Texternal = infinity) or if the combined weighted energy / enthalpy would improve due to the switch: $\langle E(before)\rangle = E(1) * p(E(1);T(1)) + E(2) * p(E(2);T(2)) > \langle E(after)\rangle = E(1) * p(E(1);T(2)) + E(2) * p(E(2);T(1))$. This is like a quench-like solution (Texternal = 0). Of course, one can also implement some external control 'temperature' Texternal and accept the switch according to some Metropolis-like rule; this option is not yet implemented.

We note that we can run all the walkers with varying temperatures (either taken sequentially from an array of variable according to some rule like in simulated annealing).

For the different walkers, the only direct interaction is the penalty for getting too close (if this penalty term is activated). Else, we just need the walkers to interact indirectly via exchange of temperatures and pressures.

Currently, one should not use more than 999 walkers (corresponding essentially to maximally 999 different pressures). Since each walker has in principle its own temperature and pressure, then 'nwalk' should in practice not exceed 'pressmax' and 'tempmax' (which must also be equal in parallel tempering). This can be different in the Monte-Carlo runs described earlier.

7.11.1 Method description

The command call for the Parallel Tempering run is: command(..) = 'mwi_partemp'. Of course, only interacting walkers can be involved and parallel tempering with only one walker is rather pointless. No parameters are set when using this command, i.e., all Parallel Tempering runs will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a Parallel Tempering run is 'mwipt'.

7.11.2 Entries of parameters (in input.f)

- 'ptTswitchflag' and 'ptPswitchflag' control whether we switch between temperatures / pressures, and which value Texternal is to have when evaluating the switching probability

    1. 'ptTswitchflag' = 0: Switch between temperatures; Texternal = infinity
    2. 'ptTswitchflag' = 1: Switch between temperatures; Texternal = 0.
    3. 'ptTswitchflag' = 2: Do not switch temperatures at all

We proceed analogously for the pressures:

1. 'ptPswitchflag' = 0: Switch pressures; Texternal = infinity, i.e. all switches are accepted
2. 'ptPswitchflag' = 1: Switch pressures: Texternal = 0, i.e. only improvements in weighted enthalpy are accepted.
3. 'ptPswitchflag' = 2. Do not switch between pressures

Note that setting both 'ptTswitchflag' and 'ptPswitchflag' to 2, results in only a bunch of Monte-Carlo simulations at various constant temperatures and pressures.

There are many combinations and schedules one can envision. If we want to perform e.g. a regular simulated annealing with pressure switches along the way, then select 'ptTswitchflagT' = 2, with all initial temperatures and temperature decrease parameters identical, and different exchangeable pressures for all the walkers.

```
ptTswitchflag = 1
ptPswitchflag = 1
```

- 'ptintmax' is the number of intervals employed for a given set of temperatures and pressures. After each interval, a switch of temperatures and pressures occurs. 'ptintmax' should be smaller than 10000. The length of each interval is 'ptintlength'.

```
ptintmax = 100
ptintlength = 1000
```

- We adjust the temperature program with 'ptflagT', analogously to the Monte Carlo runs.

  1. 'ptflagT' = 0: Read from an array
  2. 'ptflagT' = 1: T = ptTinit*(ptdecr)**ptloop
  3. 'ptflagT' = 2: T = ptTinit/(ptloop*ptdecr+1.0d0)
  4. 'ptflagT' = 3: T = ptTinit - ptloop*ptdecr

  ```
  ptflagT = 0
  ```

  The number of different starting sets of temperatures+pressures is given by a) 'ptloopmax', b) 'ptTlimit', c) 'ptnumTarray'. Unless we pick the temperatures from a pre-defined array, called 'ptTarray', we stop if either 'ptloopmax' or 'ptTlimit' has been reached. If we pick from an array with 'ptnumTarray' entries, we stop once these have been run through. 'ptloopmax' must be smaller than 1000; 'ptnumTarray' must be smaller or equal tempmax. We start counting with the internal counter 'ptloop' = 0, unless we read from the temperature list when we start with 'ptloop' = 1. The starting temperature is 'ptTinit',

unless we read from the temperature list.

```
ptloopmax = 2
ptTlimit = 0.0001d0
ptnumTarray = 2
```

For the array with eight walkers and two temperatures per walker, we would have entries like

```
do i20 = 0,nwalkmax
do i1 = 1,ptnumTarray
ptTarray(i20,i1) = 1.0d4
enddo
ptdecr(i20) = 0.05d0
ptTinit(i20) = 1.0d0
enddo
ptTarray(1,1) = 1.0d0
ptTarray(2,1) = 0.9d0
...
ptTarray(8,2) = 0.25d0
```

The entries for the walker 0 are not relevant in the application envisioned.

Note that every walker will have its own variable temperature schedule. The idea is that we can e.g. slowly decrease all temperatures at some rate, while still performing our parallel tempering temperature switches, of course. Of course, 'ptTinit' needs to be different for different walkers (if we switch temperatures) if we do not choose from an array.

- The pressure is always read from the pressure list, and we must make sure that 'pressmax' is large enough to keep in step with the number of different temperature sets! If we do not want to change the pressures, then we set 'ptPswitchflag' = 2. If 'ptflagT' > 0, then we may use different values for 'tempmax' and 'pressmax' in defining array sizes for 'ptTarray' and 'ptParray' (and also different values for 'ptnumTarray' and 'ptnumParray'), in principle.

  Note that we need to enter these data; however, the pressure(s) might be unchanged from step to step. Also, if we want to run only an exchange of temperatures, with the same pressure everywhere, then set 'ptPswitchflag' = 2, and give all the pressures the same value.

```
ptnumParray = ptnumTarray
do i20 = 0,nwalkmax
do i1 = 1,ptnumParray
ptParray(i20,i1) = 1.0d4
enddo
enddo
```

  The pressure array may look like (again the walker zero is irrelevant

but still needs an entry)
```
ptParray(1,1) = 1.0d-5
...
ptParray(8,2) = 0.55d-2
```

- 'ptmoveTflag' is the analogous flag to 'moveTflag' in the other types of random walker based runs. It tells us, whether to adjust the step-size according to some rule ('ptmoveTflag' = 1) or keep the values fixed ('ptmoveTflag' = 0).
```
ptmoveTflag = 0
```

- 'ptcurnum' is analogous to 'savecurnum'. Every 'ptcurnum' steps we write out the current configuration, overwriting the one written out before, in order to get some feeling where the walkers are. 'ptcurnum' = 0 means that nothing is written.
```
ptcurnum = 0
```

- 'ptwatchenergy' indicates whether the average energy and standard deviation sigma should be printed for each interval (=1) or not (=0). Currently, this option is not active
```
ptwatchenergy = 0
```

- If we want to keep track of the local best configuration in each interval and save it (and possibly the result of quenching this configuration), then we set 'ptsavelocflag' equal 1. If we want to perform a quench run for every local best configuration after each interval, we set 'ptquenchnumloc' = number of steps in the quench run. If no quench run is intended, set 'ptquenchnumloc' = 0. If we want to perform a gradient descent after the quench for each local best configuration, set 'ptgraddesclocflag' = 1.
```
ptsavelocflag = 1
ptquenchnumloc = 10
ptgraddesclocflag = 0
```

- If we want to perform periodically local minimizations along the trajectories of the walkers, set 'ptminimizationflag' equal 1. 'ptquenchloop' is the number of times one stops along the trajectory (must be smaller than 'ptquenchloopmax'). 'mpt2max' is the number of times we quench from a holding point.
```
ptminimizationflag = 1
ptquenchloop = 5
mpt2max = 5
```

- 'pttypeflag' indicates, which kind of local run-algorithm is to be used in the local minimization method for 'ptminimizationflag' = 1. INTRODUCE LOCALRUN INSTEAD

1. 'pttypeflag' = 0: standard stochastic quench
2. 'pttypeflag' = 1: gradient steepest descent
3. 'pttypeflag' = 2: simplex-method (not yet activated)
4. 'pttypeflag' = 3: line search
5. 'pttypeflag' = 4: powell-method
6. 'pttypeflag' = 5: no stepwise minimization

`pttypeflag = 0` If we want to perform a quench run for an intermediary configuration for 'pttypeflag' = 0, we set 'ptquenchnum' = number of steps in the quench run. If no quench run is intended, set 'ptquenchnum' = 0, or 'pttypeflag' =5. Using 'ptgraddescflag' = 1, one can perform a single one-step gradient descent nevertheless.
`ptquenchnum = 1000`

If we want to perform a gradient descent after the quench for each local best configuration, set 'ptgraddescflag' = 1. If one only wants a gradient descent, use 'pttypeflag' = 5 together with 'ptgraddescflag' = 1.
`ptgraddescflag = 1`

- If we want to start for each new pressure/temperature set with the same input configuration (randomly generated or loaded in before starting parallel tempering), then set 'useptinpflag' = 1). If 'usptinpflag' = 2, then we use the configuration we have saved as the 'ptbest' configuration of the particular walker. Else ('useptinpflag' = 0) we continue with the preceding configuration (usually a pretty good one for each walker from the preceding set of temperatures and pressures.
`useptinpflag = 2`

7.12 Threshold

The threshold subroutine performs a run starting from a configuration usually loaded in in the step before (typically deep-lying in energy). The length of the threshold-run is given by 'looplim' in the call of the subroutine, and the maximal allowed energy (energy lid) is given by 'Elimit' in the call of the subroutine, both in the control-program.

Along the threshold run, one can perform many local samplings of the underlying local minima. This can be either done by setting up a command structure (see section 'metascripts') or by performing every 'thqloop' threshold steps a set of 'numthq' quench runs of length 'numthqsteps' each. If no such sampling is required, we set 'thqloop' larger than 'looplim', and one only performs a quench at the end of the threshold run, according to the metascript.

Furthermore, during the threshold run, the density of states can be sampled, and the result stored in bins. Similar to the DOS below the threshold, one can also measure the DOS of attempted moves, and also register a sample of the states one visits or attempts to visit during the run.

Finally, it is always checked, whether the run stays within the energy interval from 'EMCbottom' to 'EMCtop'. In this way, one can explore particular slices of the energy landscape. When starting from an energy minimum below 'EMCbottom', this restriction is activated once the walker has entered the energy slice. Note that this slice is currently once and for all defined in input.f, and is not changed when changing the lid.

7.12.1 Method description

The command call for the threshold run is: command(..) = 'thresh(Elimit,looplim)' or command(..) = 'mws_thresh(Elimit,looplim)' for many non-interacting walkers, respectively. Two parameters are set when using this command: 'Elimit' = lid energy, up to which the walker can rise, and 'looplim' = length of the threshold run. The prefix indicating that the output files have been generated during a Monte Carlo run is 'oneth' or 'mwsth', respectively.

Interacting multi-walker runs make not much sense for the applications envisioned for threshold explorations, and are currently not implemented.

7.12.2 Entries of parameters (in input.f)

- The threshold performs runs starting from a configuration usually loaded in in the step before, and which typically is deep-lying in energy. The length of the threshold-run is given by 'looplim' in the call of the subroutine, and the maximal allowed energy (energy lid) is given by 'Elimit' in the call of the subroutine, both in the control-program.

- Note that you can also force the threshold run to be performed under the lid but only accept moves with energies that lie within a certain energy interval given by 'EMCtop' and 'EMCbottom' (< 'EMCtop'), which is defined in the input for the simulated annealing routine.

  A special feature is that we can start a threshold run outside this interval and have the actual threshold criterion only get activated once we are inside this energy interval. Such options are of interest, if we want to explore a certain energy interval in detail while starting from a minimum whose energy lies outside this interval.

- In addition, we define 'Ebottom' as the value of the energy, where the run can be stopped when an energy is found that lies below this value. If 'Ebottomflag' = 0, then the prescribed value 'Ebottom' is used, while if 'Ebottomflag' = 1, the value of the starting configuration minus a

tolerance 'tolthresh' is used.

```
Ebottomflag = 1
tolthresh = 1.0d-4
Ebottom = -10.0d0
```

- There are three options available for E < 'Ebottom', depending on the value of 'Threshcontflag':

  1. 'Threshcontflag' = 0 makes the program stop, and the final configuration is saved as 'lowEner'... This is usually employed if one wants to analyze this new configuration, minimize it and restart the threshold from this state, especially if one attempts to use the threshold run as a global minimizer starting from some high-lying minima. One hopes that the threshold will 'dig itself' deeper into the landscape this way.
  2. 'Threshcontflag' = 1 makes the run stop, and moves the program automatically to the end of the threshold run. This option is either used when we assume that for E < 'Ebottom' the energy calculation has gone haywire and we want to stop the run before it produces nonsense, or if there is already an automatic quench planned at the end of the threshold run when searching for new minima.
  3. 'Threshcontflag' = 2 continues the run, but if at the same time 'Ebottomflag' = 1, then 'Ebottom' is reset to 'Ecurrent', and the current configuration is saved as 'lowEner'... This scheme is the one most useful if we perform regular threshold runs from some low-lying local minimum, and we want to register lower energies (or course), but do not want to interrupt the regular threshold search.
  4. 'Threshcontflag' = else, ignores the fact that a lower energy has been found and just continues the run.

  ```
  Threshcontflag = 2
  ```

- Furthermore, during the threshold the DOS is sampled, and the result stored in boxes. In order to make the DOS compatible for different Lid-values and starting minima, two options are offered, controlled by 'Ebinflag'.

  1. If 'Ebinflag' = 1, bins are defined by the energy interval 'Ebinbottom' < 'Ebintop', and 'deltabin', where 'deltabin' is the width of a bin, which appears 'binfact' times in the interval 'Ebintop' - 'Ebinbottom'.

2. If 'Ebinflag' = 2, then we ignore those energies where 'Ecurrent' exceeds 'Ebintop' or -'Ebinbottom', and just not register those values. Apart from that, 'Ebinflag' = 2 is like 'Ebinflag' = 1
3. On the other hand, if 'Ebinflag' = 0, then 'Ebinbottom' = 'Ebottom', and 'Ebintop' = 'Elimit'.

In all cases, 'deltabin' is calculated from 'binfact' (inside the code 'deltabin' corresponds to the local variable 'hvdeltabin'). Thus, 'binfact' must be an integer larger than zero.

Note that if an energy is found below 'Ebinbottom', or if an accepted energy is above 'Ebintop', the program is stopped - thus one should be careful when using the option 'Ebinflag' = 0 or 'Ebinflag' = 1, if one is not sure whether states exceeding the given range might appear (due to deeper minima or accidental starting points at higher energies.

```
Ebinflag = 2
Ebinbottom = -5.0d0
Ebintop = 5.0d0
binfact = 1000
```

- 'dosstep' counts the number of steps between sampling the DOS. Thus, 'dosstep' should be an integer larger than zero. If no sampling is desired, chose 'dosstep' larger than 'looplim' or set 'Edosflag' equal 0 instead of 1.

```
Edosflag = 1
dosstep = 100
```

- Similar to the DOS below the threshold, one might want to measure the DOS of attempted moves, controlled by 'Eattflag'.

  1. If this is to be done, with every move, we set 'Eattflag' = 1.
  2. If we want to use only every 'attstep' move, we set 'Eattflag' = 2.
  3. Not checking of attempted moves occurs for 'Eattflag' = 0.

  The values for bottom and top energies are given by 'Eattbottom' and 'Eatttop'. 'attbinfact' is the analogous factor to 'binfact'.

```
Eattflag = 1
Eattbottom = -5.0d0
Eatttop = 95.0d0
attbinfact = 1000
attstep = 1
```

- In addition to the DOS, one might want to register the configurations along the threshold run, and those that one has only attempted to reach. In the former case, one can use the savetrajectory scheme, while

for the latter one, one sets 'thconfsaveflag' = 1, and one will register every 'thconfstep' moves the attempted configuration.

```
thconfsaveflag = 0
thconfstep = 100
```

- Along the threshold run, one might want to perform many local samplings of the underlying local minima. This can be either done by setting up a command structure (see section 'metascript') or by performing every 'thqloop' threshold steps a set of 'numthq' quench runs of length 'numthqsteps' each. If no such sampling is required, set 'thqloop' larger than 'looplim'.For the quench runs, we also need a temperature update with starting value 'Tmthinit'.

  ```
  do i1 = 0,nwalkmax
  thqloop(i1) = 900000
  numthq(i1) = 2
  numthqsteps(i1) = 1000
  Tmthinit(i1) = 1.0d0
  enddo
  ```

  The type of quench routine is analogous to the options in the multi-quench procedure according to the value of 'mthtypeflag': USE LOCALRUN INSTEAD

  1. 'mthtypeflag' = 0: standard stochastic quench
  2. 'mthtypeflag' = 1: gradient steepest descent
  3. 'mthtypeflag' = 2: simplex-method (not yet activated)
  4. 'mthtypeflag' = 3: line search
  5. 'mthtypeflag' = 4: no quench is run (usually one runs a gradient descent in one step instead, i.e. 'mthgraddescflag' = 1). Else, nothing happens for 'mthtypeflag' = 4!

  ```
  mthtypeflag = 0
  ```

  If we want to run a gradient descent in one step after the quench, set 'mthgraddescflag' = 1.

  ```
  mthgraddescflag = 0
  ```

7.13 (Simulated Annealing) Local (shadow) run

The simulated-annealing-with-local-run-option allows a defined number of sequences of local runs (e.g. quench or Monte Carlo) alternating with random walks (free or at some temperature). We call this a shadow run procedure, because after several random walk steps between two reference configurations, we generate a shadow configuration using a local run, and then can e.g. compare the energies of the outcome of the shadow runs, in order to

decide, whether we accept the move from one shadow configuration to the next. One uses two different moveclasses for the random walk (no. 1) and the local run (no. 2).

7.13.1 Method description

The command call for the shadow (local) run is: command(..) = 'salocrun' or command(..) = 'mws_salocrun' or command(..) = 'mwi_salocrun' for single walker, many separate walkers and many interacting walkers, respectively. No parameters are set when using this command, i.e., all shadow runs will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a shadow(local) run is 'onelr', 'mwslr', or 'mwilr'. Note that there can be many intermediary configurations (outcomes of the local quenches) that are saved, too, indicated by the prefix 'imlr'. The local minimizations take place in the module 'Local Run'. Such a minimization / local run can be a stochastic quench, gradient, line search, simplex, MonteCarlo-run, graddesc or powell routine. Also, we can perform a local run and follow it up with a graddesc. The special feature is that moves are accepted not based on their current energy but based on the energies of their 'shadow' configuration, i.e. the outcome of a minimization or a local run after the actual move.

Finally, we can perform a second minimization, using a different method, moveclass and/or energy function.

7.13.2 Entries of parameters (in input.f)

Note: If not specifically defined here, the values in the local run called by salocrun are the ones given in the default local-run parameters (see subsection 'Local Run' below).

- The Simulated Annealing Local Run-option allows a defined number of local run (e.g. quench or Monte Carlo) + random walk (free or at some temperature) cycles, where the number of steps and temperature for each local run ('salrnumsteps', T = 0 or T = 'Tlocal') and random walk (of length 'lrtnum', T > 'Tinf' or T = 'Tlr') is prescribed, according to the flag 'lrTinfflag':

  1. If we want to use an infinite temperature for the random walk, set 'lrTinfflag' = 1.
  2. For a finite temperature 'Tlr', 'lrTinfflag' = 0.

  The currently envisaged application is with 'lrtnum' = 1 and 'lrTinfflag' = 1, i.e. T(random walk) > 'Tinf'.

  ```
  lrtnum = 1
  lrTinfflag = 1
  ```

- Analogous to the multi-quench procedure, one block corresponds to 'lrnum' local runs (of length 'salrnumsteps' where applicable) and 'lrnum'-1 Monte Carlo runs of length 'lrtnum' at temperature 'Tlr' - given adaptively or with exponential decrease -, or at temperature 'Tinf' (also called heating runs). One such sequence defines a block.
  `lrnum = 10`

- 'lrloop' counts the number of blocks (run at constant temperature).
  `lrloop = 1`

  Note that 'lrnum' and 'lrloop' should be smaller than 'lrloopmax' and 'lrnummax' set in 'varcom.inc'.

- Temperature control of the random walk in the heating phase

  1. 'Tlr0' is the initial temperature for the heating in the shadow run subroutine, and Tlr(n) = Tlr(n-1)*('flr').
  2. If we want free random walks for the heating, we should set 'Tlr0' equal to 'Tinf' (defined below), and 'flr' = 1.0d0. Alternatively, we use 'lrTinfflag' = 1.
  3. If the initial temperature 'Tlr0' is to be adjusted automatically, one should set 'Ttest' to some reasonable starting value, and 'init5flag' = 1 (see description of initial temperature initialization in simulated annealing routine).
  4. 'Tlrinit' is the starting temperature of each of the quench runs in the local run subroutine (no longer active? CHECK).

```
flr = 1.0d0
do i20 = 1,nwalkmax
Tlr0(i20) = 1.0d9
Tlrinit(i20) = 1.0d0
enddo
```

- Temperature control of random walks inside the local run called by salocrun.

  1. 'Tlocalsalr0' is the starting temperature of the local run Monte Carlo walks (MC and short simann runs, instead of a quench) in 'salocalrun'.
  2. The local temperature can be adjusted for each loop by a decreasing factor 'fsalrlocal': Tlocal(n) = Tlocal(n-1)*'fsalrocal'.

```
fsalrlocal = 1.0d0
do i20 = 1,nwalkmax
Tlocalsalr0(i20) = 1.0d0
enddo
```

- 'lrflag1 controls how the temperature of the heating phase should be adjusted after each block

  1. 'lrflag1' = 1: automatic temperature decrease after each block
  2. 'lrflag1' = 0: corresponds to adaptive schedule (not active)

  `lrflag1 = 1`

- 'lrflag2' controls the saving of the final configuration of a local run within a block

  1. 'lrflag2' = 1: save final configuration of each block, plus the writing of its energy into the protocol
  2. 'lrflag2' = 0: no recording.

  `lrflag2 = 1`

- 'lrflag3' controls the recording of rejection/acceptance of moves for statistical purposes

  1. 'lrflag3' = 1: Yes
  2. 'lrflag3' = 0 No

  `lrflag3 = 0`

- 'salrtypeflag' indicates, which kind of local run-algorithm is to be used in the sa-local-run-method.

  1. 'salrtypeflag' = 0: standard stochastic quench
  2. 'salrtypeflag' = 1: gradient steepest descent
  3. 'salrtypeflag' = 2: simplex-method (not yet activated)
  4. 'salrtypeflag' = 3: line search
  5. 'salrtypeflag' = 4: powell-method
  6. 'salrtypeflag' = 5: Monte Carlo at temperature 'Tlocal'
  7. 'salrtypeflag' = 6: gradient steepest descent in one step (e.g. with GULP)
  8. 'salrtypeflag' = 7: quench followed by gradient descent in one step
  9. 'salrtypeflag' = 8: MC at 'Tlocal' followed by gradient descent in one step

  `salrtypeflag = 0`

  ADD PARAMETERS FOR SECOND MINIMIZATION The cut-off criterion for a local run in salocrun is given by 'salrnumsteps'. Adaptive halting criteria for MC/SA are not implemented; flag for adaptive is 'salrnumsteps' = 0)
  `salrnumsteps = 1000`

7.14 Local Run

In many modules, we want to perform a number of local runs (Quench, Monte Carlo, Gradient descent, perhaps a short simann run) as part of a larger simulation. Thus there exists a separate subroutine localrun, which is called every time a local run is to be performed. Note that this subroutine cannot be called by the used via one of the commands in the command-script: this subroutine is only called from some other module, named the 'calling module', or the moveclass (if a local optimization is part of the move, as often happens when using jumpmoves or multiwalkermoves. In this calling module, we usually can define the values of the parameters for the 'local run', which then replace the default values defined here.

Typically, a local run can be a stochastic quench, gradient, line search, simplex, graddesc or powell routine. Also, we can run a quench and Monte Carlo (or simulated annealing), and follow it up with a graddesc.

If we want to perform two (different) local runs one after the other, with the possibly saving the intermediary configuration, i.e. the outcome of the first local run, we set 'localrun2flag' to one. This is usually done in the calling module, 'localrun2flag0' is just the default value. We also set the current value of 'localrun2flag' equal to 'localrun2flag0

```
localrun2flag0 = 0
localrun2flag = localrun2flag0
```

Furhermore, we have: number of steps for each stochastic local run (MC, SA, Quench): 'lrnumsteps'.
Temperature for MC/SA: 'Tlocal'
lrnumsteps0 is the default value for 'lrnumsteps; usually each calling program overrides this value according to its own definitions. The same applies to the length of the second minimization that can be performed in local run, 'lrnumsteps10'. Again, we set the current value of these parameters equal to the value defined in localrun

```
lrnumsteps0 = 1000
lrnumsteps = lrnumsteps0
lrnumsteps10 = 10
lrnumsteps1 = lrnumsteps10

   We record the energy every 'wrsteploc' local-run steps. If 'watch'
= 1, then we also save the configuration.
wrsteploc = 10000000
   'Tlocal0' is the starting temperature of the local run Monte
Carlo walks. It can be adjusted for each loop by a decreasing factor
'flrlocal': 'Tlocal(n)' = 'Tlocal(n-1)'*'flrlocal'. Usually, these
values are overridden by the subroutine calling localrun. Note that
currently the localrun-parts always assume that a constant temperature
MC-run is performed, and not an annealing run, i.e. we set 'f??local'
```

= 1.0d0 automatically. Analogously, 'Tlocal10' is the corresponding temperature for the second minimization within the local-run call.

```
do i20 = 0,nwalkmax
Tlocal0(i20) = 1.0d0
Tlocal(i20) = Tlocal0(i20)
Tlocal10(i20) = 1.0d0
Tlocal1(i20) = Tlocal10(i20)
enddo
flrlocal0 = 1.0d0
flrlocal = flrlocal0
```

'lrtypeflag0' indicates, which kind of local run-algorithm is to be used in the localrun-subroutine as default. This is usually explicitly defined within the calling subroutine. Similarly, 'lrtypeflag10' indicates the type of local run for the second minimization that calls the localrun.

- 'lrtypeflag' = 0: standard stochastic quench
- 'lrtypeflag' = 1: gradient steepest descent
- 'lrtypeflag' = 2: simplex-method (not yet activated)
- 'lrtypeflag' = 3: line search
- 'lrtypeflag' = 4: powell-method
- 'lrtypeflag' = 5: Monte Carlo at temperature 'Tlocal'
- 'lrtypeflag' = 6: gradient steepest descent in one step (e.g. with GULP)
- 'lrtypeflag' = 7: quench followed by gradient descent in one step
- 'lrtypeflag' = 8: MC at 'Tlocal' followed by gradient descent in one step

```
lrtypeflag0 = 6
lrtypeflag = lrtypeflag0
lrtypeflag10 = 0
lrtypeflag1 = lrtypeflag10
```

If we want to save the output of every localrun minimization only after the first minimizer, set 'localrunsave' = 1; only after the second, set 'localrunsave' = 2; after both, set 'localrunsave' = 3. Recall that 'poolmincount' already takes care of the saving of minima after 'jumpmoves' and 'multiwalker moves', so one does

not need to save them a second time in localrun itself. Up to 99999
such saves can be performed for each run with a different 'confname'.
localrunsave = 0
   dsteplr0 is the (initial) step-size of the gradient in local
run (needed for built-in gradient and line search option).
dsteplr0 = 0.01
   'climitlr is the number of downhill steps for a given gradient
in the line search in local run.
climitlr = 50

7.15 Thermal cycling / multi quench

Thermal cycling / multi-quench is a special case of the simulated
annealing/salocrun procedure. It allows a defined number of quench+random-walk
sequences, where the quench can be a stochastic quench, gradient,
line search, simplex or the powell routine. During the random walk,
steps are accepted either always (T = 'Tinf') or according to simulated
annealing at some temperature which is reduced according to a schedule.
In contrast to salocrun, we continue our search from the local minimum
we have reached and do not jump back to the previous local minimum
if the current one has higher energy than the previous one.

7.15.1 Method description

The command call for the multi-quench is: command(..) = 'multiquench'
or command(..) = 'mws_multiquench' or command(..) = 'mwi_multiquench'.
No parameters are set when using this command, i.e., all multi-quench
runs will follow the set of parameters defined in the input to G42+.
The prefix indicating that the output files have been generated
during a multi-quench run is 'onemq', 'mwsmq' or 'mwimq'. Note that
there can be many intermediary configurations (outcomes of the local
quenches) that are saved, too, indicated by the prefix 'im'. Note
that for interacting multiple walkers, we only take the interaction
into account when we perform the high-temperature phase of the walk,
and not during the local minimizations.

7.15.2 Entries of parameters (in input.f)

The MQ-option allows a defined number of minimization+random walk
cycles, where the number of steps for each minimization ('mqnumsteps')
and random walk ('tnum') at temperature Tmq is prescribed.

- 'mqloop' counts the number of blocks (each bloack is run at
  a constant temperature 'Tmq', but the temperature can be varied
  from block to block - adaptively or with exponential decrease).

```
'mqloop' should be smaller than 'mqloopmax' (defined in varcom.inc)
mqloop = 20
```

- One block corresponds to 'mqnum' quenches and 'mqnum'-1 heatings at temperature 'Tmq' - given adaptively or with exponential decrease. One such sequence defines a block. 'mqnum' should be smaller than 'mqnummax' (set in 'varcom.inc').
  mqnum = 40

- Number of steps for each random walk at the heating stage is 'tnum'.
  tnum = 1000

- 'Tmq0' is the initial temperature for the heating in the multiquench subroutine, and Tmq(n) = Tmq(n-1)*('fmq'). If we want free random walks for the heating, we should set Tmq0 larger or equal to 'Tinf' (defined below), and 'fmq' = 1.0d0. If the initial temperature 'Tmq0' is to be adjusted automatically, one should set 'Ttest' to some reasonable starting value, and 'init5flag' = 1 (see below; CHECK whether still active!). 'Tmqinit' is the starting temperature of each of the quench runs in the multi-quench subroutine. 'fmqlocal' is the decrease exponent for the MonteCarlo-local run called from multiquench, and 'Tlocalmq0' the starting temperature.

```
fmq = 0.95d0
fmqlocal = 1.0d0
do i20 = 1,nwalkmax
Tmq0(i20) = 1.0d0
Tmqinit(i20) = 1.0d0
Tlocalmq0(i20) = 1.0d0
enddo
```

- 'mqflag1' controls the change of the temperature between blocks:

  1. 'mqflag1' = 1 corresponds to T-decrease after each block
  2. 'mqflag1' = 0 corresponds to adaptive schedule (not active).

  mqflag1 = 1

- 'mqflag2' controls the saving of the final configuration of a quench-run with a block in the MQ-run, plus the writing of the energy into the protocol

  1. 'mqflag2' = 1: save and record
  2. mqflag2 = 0: no saving/recording.

  mqflag2 = 1

- 'mqflag3' controls the recording of rejection/acceptance of moves

  1. 'mqflag3' = 1: recording
  2. 'mqflag3' = 0: no recording.

  mqflag3 = 0

- 'mqtypeflag' indicates, which kind of quench-algorithm is to be used in the multiquench-method when calling the Localrun subroutine.

  1. 'mqtypeflag' = 0: standard stochastic quench
  2. 'mqtypeflag' = 1: gradient steepest descent
  3. 'mqtypeflag' = 2: simplex-method (not yet activated)
  4. 'mqtypeflag' = 3: line search
  5. 'mqtypeflag' = 4: powell-method
  6. 'mqtypeflag' = 5: Monte Carlo at temperature Tlocal
  7. 'mqtypeflag' = 6: graddesc in one step
  8. 'mqtypeflag' = 7: quench plus graddesc in one step
  9. 'mqtypeflag' = 8: MC plus graddesc at the end

  mqtypeflag = 0 ADD SECOND MINIMIZATION PARAMETERS

- The following are cut-off criteria for the various kinds of quench algorithms that can be used in MQ

  1. Cut-off criterion for quench inside the multi-quench is given by 'mqnumsteps' (adaptive halting criteria not implemented; flag for adaptive is 'mqnumsteps' = 0)
     do i20 = 0,nwalkmax
     mqnumsteps(i20) = 1000
     enddo
  2. 'dstepmq0' is the (initial) step-size of the gradient in multiquench (needed for gradient and line search option)
     dstepmq0 = 0.01
  3. 'climitmq' is the number of downhill steps for a given gradient in the line search in multiquench
     climitmq = 50

7.16 Prescribed path

The prescribed path option permits a number of steps along a well-defined route or within a well-defined set of possible moves (random walk at infinite or finite temperature, usually for a highly restricted moveclass), which is usually followed by a local minimization or short exploration starting from these points (e.g. a stochastic quench, low-temperature Monte Carlo, or Powell minimization), in order to explore specific parts or aspects of the energy landscape. Such a route can constitute an attempt to move from one minimum to another (exploring e.g. a phase transition if both modifications involved are minima on the same landscape) where these minima do not have to be neighbors. In that case, the exploration of the neighborhood of the path on the landscape can give us information about important side-minima that the system might encounter during the actual transition (the first route proposed by e.g. defining a starting and and a target structure is usually an unrealistic path on the landscape).

The currently envisaged application is with a quench after each step along the path, and T (random walk at restricted moveclass) $>$ 'Tinf'. Using a finite temperature for the restricted random walk is not really useful for the currently implemented set of paths - if you want to do e.g. a real simulated annealing with a restricted moveclass, then it makes sense to set this up as a standard annealing run instead of a prescribed path run.

Note that the local run is performed with moveclass no. 2, while the prescribed path exploration uses moveclass no. 1. Furthermore, for the stochastic random walks, these local runs can be restricted to the part of the landscape that is orthogonal to the prescribed path (if such an orthogonality can be defined and makes sense).

7.16.1 Method description

The command call for the prescribed path is: command(..) = 'path' or command(..) = 'mws_path' for many independent walkers, respectively. No parameters are set when using this command, i.e., all path runs will follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a prescribed path run is 'onepa' or 'mwspa'. Note that there can be many intermediary best or worst configurations (outcomes of the local quenches) that are saved, too, indicated by the prefix 'impabe' or 'impawo', respectively.

7.16.2 Entries of parameters (in input.f)

- Temperature control

1. Number of steps and temperature ('patnum', T > Tinf or
   T = Tpa) when proceeding along the prescribed path (if
   a random walk with a restricted moveclass is envisioned)
   before the local quench takes place. The currently envisaged
   application is with 'patnum' = 1 (quench after each step)
   and T (random walk) > Tinf. If we want to use an infinite
   temperature for the random walk, 'paTinfflag' = 1. For
   a finite temperature 'Tpa', 'paTinfflag' = 0. This second
   option is currently not implemented.
   patnum = 1
   paTinfflag = 1
2. For 'paTinfflag' = 0, 'Tpa0' is the initial temperature
   for the heating in the prescribed path subroutine, and
   Tpa(n) = Tpa(n-1)*('fpa'). Note that this option currently
   is not active! If we want free random walks for the prescribed
   path random walk, we should set 'Tpa0' equal to 'Tinf'
   (defined in simulated annealing), and 'fpa' = 1.0d0. Alternatively,
   we use 'paTinfflag' = 1. If the initial temperature 'Tpa0'
   is to be adjusted automatically, one should set 'Ttest'
   to some reasonable starting value, and 'init5flag' = 1
   (see simulated annealing).
   Tpa0 = 1.0d9
   fpa = 1.0d0

- Somewhat analogous to the multi-quench, the prescribed path
  module proceeds via blocks or runs. Each block corresponds
  to 'panum' local runs of length 'patnum'. After each local
  run (for fixed pre-defined paths this is usually only one step),
  a local relaxation or exploration follows of length 'panumsteps'
  (if applicable). One such sequence defines a block. The product
  of 'panum' and 'patnum' (usually equal 1) must be equal to
  the total number of steps of the path, 'Npath' = 'panum' *
  'patnum'! For special cases, one can choose not to fulfill
  this condition, in order to go only a few steps along the full
  path (make sure that 'Npath' $\leq$ 'palengthmax' given in 'varcom.inc').
  Make sure to set 'panum' = number of possible permutations,
  and 'patnum' = 1, if we test completely for all configurations
  in alloys that are generated via atom exchange with 'pathflag'
  = 4.
  Npath = 100
  panum = 100

- Block-structure of prescribed paths (currently only one block
  is active: 'paloop' = 1, always)

1. 'paloop' counts the number of blocks (run at constant temperature).
   Currently, there is no point in using values different
   from 'paloop'=1. The algorithmic block-based structure
   analogous to the multi-quench is kept, in order to be able
   to run more complex paths in a nested fashion (make sure
   that 'paloop' $\leq$ 'paloopmax' given in 'varcom.inc')
   paloop = 1
2. 'paflag1' = 1 corresponds to T-decrease after each block;
   'paflag1' = 0 corresponds to adaptive schedule (not active).
   paflag1 = 1
3. 'Tpainit' is the starting pseudo-temperature of each of
   the quench runs in the block-structure prescribed paths.
   Do not forget to check 'quTdecrease' etc. in the quench
   routine.
   Tpainit = 1.0d0
4. 'Tlocal0pa' is the starting temperature of the block-structure
   Monte Carlo walks. It can be adjusted for each loop by
   a decreasing factor 'fpalocal': Tlocal(n) = Tlocal(n-1)
   * 'fpalocal'. This option is not active at the moment,
   since such a block-type prescribed path is not implemented.
   Tlocal0pa = 1.0d0
   fpalocal = 1.0d0
5. 'paflag2' = 1 corresponds to the saving of the final configuration
   of a local run within a block in the sub-run, plus the
   writing of its energy into the protocol. 'paflag2' = 0
   means no recording.
   paflag2 = 1
6. 'paflag3' = 1 means recording of rejection/acceptance of
   moves; 'paflag3' = 0 means no recording.
   paflag3 = 0
7. 'paflag4' = 1 means saving every patnum steps the non-relaxed
   configuration along the path; 'paflag4' = 0 means no saving.
   paflag4 = 1

- Control of the local runs

1. 'patypeflag' indicates, which kind of local run-algorithm
   is to be used in the prescribed path-method.
   'patypeflag' = 0: standard stochastic quench;
   'patypeflag' = 1: gradient steepest descent;
   'patypeflag' = 2: simplex-method (not yet activated);
   'patypeflag' = 3: line search;
   'patypeflag' = 4: powell-method;

'patypeflag' = 5: Monte Carlo at temperature 'Tlocal'
'patypeflag' = 6: One step gradient descent
'patypeflag' = 7: quench followed by one step gradient descent
patypeflag = 0 ADD LOCALRUN OPTION??

2. If 'paorthoflag' = 1, we only allow relaxation/MC moves orthogonal to the path. No orthogonal gradient-descent is currently implemented.
   paorthoflag = 0
3. In addition, we have the option to set a specific local moveclass just for the path minimizations (which then could be different from the outcome of a regular minimization with the moveclass 'mclflag_local'). If we set 'pamclflag_local' = 0, then we use the normal moveclass 'mclflag_local' (which is going to the case for most situations, and for which the default is 2).
   pamclflag_local = 0
4. Cut-off criterion for local run in prescribed run is given by 'panumsteps' (adaptive halting criterion for quenches is not implemented; flag for adaptive is 'panumsteps' = 0).
   panumsteps = 30000
5. 'dsteppa0' is the (initial) step-size of the gradient in local run (needed for gradient and line search option).
   dsteppa0 = 0.01
6. 'climitpa' is the number of downhill steps for a given gradient in the line search in local run.
   climitpa = 50

- There are currently several types of prescribed-paths that have been programmed for this module; in principle, these can be extended, and additional paths be designed.

  The variable 'pathflag' denotes which kind of path is being prescribed. 'pathlist(i)' lists the numbers of the building units, with i = 1,'Npathlist', which are to be changed along the path.

  1. 'pathflag' = 1: Rotation about an axis of a building unit. Here, 'pathangleflag' indicates the axis (1 = z, 2 = x, 3 = y). 'startangle' and 'endangle' are the starting and end angle of the rotation, with 'Npath' equidistant steps. 'pathlist(1)' is the number of the building unit that is

to be rotated. Make sure it is at the right position in
the starting configuration.
pathangleflag = 1
startangle = 0.0d0
endangle = 2.0d0*pi

2. 'pathflag' = 2: Translation of one or several building
units from their starting values to target values 'xtarget',
with 'Npath' equidistant steps.
Note that the first index of 'xtarget' gives the position
in the actual configuration (and not the position in the
'pathlist'), i.e., the first entry in the path list might
be building unit no. 4, and thus the first target has the
coordinates 'xtarget(4,..)' = ...
xtarget(1,1) = 0.0d0
xtarget(1,2) = 0.0d0
xtarget(1,3) = 0.0d0

3. 'pathflag' = 3: Modification of the three cell vectors
from the initial values to target values 'atarget', with
'Npath' equidistant steps.
atarget(1,1) = 0.0d0
atarget(1,2) = 0.0d0
atarget(1,3) = 0.0d0
atarget(2,1) = 0.0d0
atarget(2,2) = 0.0d0
atarget(2,3) = 0.0d0
atarget(3,1) = 0.0d0
atarget(3,2) = 0.0d0
atarget(3,3) = 0.0d0

4. 'pathflag' = 4: Exchange of atoms systematically or at
random.
Here, setting 'pathflag4' = 1 means systematic exchange
(possible for a small number of feasible exchanges, and
'pathflag4' = 0 means at random (better for large numbers
of exchanges, and required for the exchange of more than
two numbers of atoms that are involved in the exchange.
pathflag4 = 0
The program takes the atoms that are allowed to be exchanged
according to the values of 'fixbg' for moveclass number
1 (or the currently set moveclass for the global module).
You need to give the total number of atoms affected ('Npathlist')
beforehand for a check.
The entries in 'pathlist' are overwritten at the beginning
from the path-module before the generation of new configurations

takes place. You should also, when attempting a systematic generation of permutations, set 'panum' equal to that number and make sure it is < 'Npath', and 'patnum' = 1. Note that, currently, we only can systematically permute two types of atoms. For more building unit types, we have to use the random option

5. 'pathflag' = 5: Make random steps according to a restrictive moveclass, always starting from the same reference configuration. The flag 'pathflag5' = 1 indicates that only one building unit is affected in the random walk (the first one in the 'pathlist'). Else one should set this value to zero. So far, we have not implemented the option that a selected group (> 1) of building units (given in 'pathlist') can be moved during the random walk.

6. 'pathflag' = 6: Move between the starting configuration and an externally given target configuration (name: 'pathtarget'//confname/ in equidistant steps. In this case, 'pathlist' must include all atoms, i.e. 'Npathlist' = 'nbgcur'. Note that every walker 'hvwalk' can have its own target configuration if desired. pathflag5 = 1

7. Other values of 'pathflag' correspond to more complicated paths, which are not yet implemented.
   pathflag = 1

Npathlist = 1
pathlist(1) = 1

Note that 'Npathlist', 'pathlist', 'xtarget', 'atarget', 'startangle', 'endangle' and 'Tlocalpa0' all have an additional index giving the walker under consideration, since each walker can have its own special path ('pathflag' is the same for all walkers, however).

7.17 Find periodic sub-cell of the simulation cell

The subroutine findcell can be used to determine whether parts (or the whole) of a configuration exhibit approximately translational symmetry.

7.17.1 Method description

The command call for findcell is: command(..) = 'findcell'. No parameters are set when using this command, i.e., all findcell searches will

follow the set of parameters defined in the input to G42+. The prefix indicating that the output files have been generated during a findcell search is 'onefc' or 'mwsfc'. The output appears in the protocol. Not yet implemented are subroutines that generate actual configuration files for the possible candidates ready for plotting in the output in the protocol,

7.17.2 Entries of parameters (in input.f)

All the parameter refer to the acceptance criterion of the test-cells. Note that only test cells are considered whose volume is less than the one of the original cell (else one should directly proceed with the further analysis using the tools in KPLOT).

- 'fctol1' gives the sine of the angle between the vectors of the test cell.
  fctol1 = dsin(pi/6.0d0)
- 'fctol2' gives the latitude for hitting atoms of the shifted test cell.
  fctol2 = 0.1d0
- 'fctol3' gives the fractional shift of the testcell off the aufpunktposition.
  fctol3 = 0.95d0
- 'fctol4' gives the volume difference necessary for calling two subcells similar enough to force a comparison analysis. Since this needs to be computed according to
  if (dim.eq.3) then
  fctol4 = fctol4_0*(4d0*pi/3.0d0)*(radbest)**3
  elseif (dim.eq.2) then
  fctol4 = fctol4_0*pi*(radbest)**3
  else
  fctol4 = fctol4_0*radbest
  endif,
  we just assign a value to 'fctol4_0'.
  fctol4_0 = 0.5d0
- 'fctol5' gives the volume fraction of the test cell.
  fctol5 = 0.55d0
- 'fctol6' gives the flatness of the test cell.
  fctol6 = 0.2d0
- 'countmin' gives the minimum number of repetitions of the testcell needed for acceptance. Recommended values for three, two and

one dimensions are 4, 3 and 2, respectively.
countmin = 4

7.18 Setting the pressure

There are two ways, the pressure can be set in G42+. There is the explicit entry 'pressure = ...' in the input.f file. Furthermore, we can use the 'setpress' command to reset the pressure to a new value.

7.18.1 Method description

The command for setting a new pressure is command(..) = 'setpress (real,integer)'. There are two parameters required, of which the second one is currently a dummy variable (might be used as a counter, but is not yet activated).

7.18.2 Entries of parameters (in input.f)

- 'press' is the external pressure (real number in eV/Ang**3; 1eV/Ang**3 = 1.6 x 10**11 J/m**3 = 1.6 x 10**6 atm)
  press = 0.0d0
- 'setpress (real,integer)' is the command to set the pressure to a new value. 'real' represents a real variable, which must contain only numbers, a dot and a -sign; no letters are allowed. 'integer' is a dummy variable not yet activated (e.g. set equal to 1).

  When using the metascript to run through many different pressures, it is useful to employ an array called 'hvpress', where all the pressures to be used are listed. The number of different pressures is then given by 'hv1press'. Alternatively, in a special scenario of running through many different pressures starting from a certain configuration one uses 'npress' as a counter. For more details see the metascript section.
  setpress(0.00000,1)
  hvpress(1) = '0.00000000000000'
  hv1press = 1
  npress = 0

7.19 Random seed

The random seed generator is initialized by assigning a value to the variable 'seed', which is the starting value for the random

number generator. There are two ways to do this: directly in the input file, and by using the 'setseed'-command.

7.19.1 Method description

The command for setting a new random seed is command(..) = 'setseed(i)', which initializes the random number generator with the seed 'i' (i must be an integer larger than 0). This seed is then used as the current masterseed, from which the individual seeds of the random walkers inside the mws-type runs are derived: seed(iwalk) = masterseed + iwalk, or seed = masterseed for single walker modules. For purposes of reproducibility, it is recommended to explicitly use the 'setseed' command before calling a module that employs random walks. Note that in mws-modules that do not employ random numbers (gd, ls, po, sx, fc) the seed is not changed.

7.19.2 Entries of parameters (in input.f)

- 'seed' is the initial starting value for the random number generator.
  seed = 1
- 'setseed(i)' is the command to set the seed to a new value. The new seed i must be an integer.
- There are a number of instances when using metascripts that variables are defined that have the meaning of a seed. The seed-value becomes often part of the name of configuration files, in order to keep track of themWhen blocks of commands are repeated for the different seed values, then the first seed variable is 'hvstart'+1.
  hvstart = 0000
- For threshold runs, 'nthseed' is the number of seeds for a given lid value 'thseedinit' is the value of the seed at the beginning of the generation loop. (analagous to 'hvstart') Note that we always generate a new seed value at the beginning of another threshold. In this way, we can reproduce pieces of the threshold analytically if desired.
  nthseed = 30
  thseedinit = 1

7.20 Set initial number of atoms

The variable 'natominit' describes the initial number of atoms in a configuration. It is meant to allow the user to load atomconfigurations

with different numbers of atoms, not even necessarily with the same stoichiometry. In its current form, the command is obsolete. There are two ways to set the initial number of atoms: directly in the input file, and by using the setnatominit-command.

7.20.1 Method description

The command for setting a new random seed is command(..) = 'setnatominit(i)'. Here, the integer i constitutes the number of atoms initially present in the configuration.

7.20.2 Entries of parameters (in input.f)

The current standard way to enter the number of atoms is via the entry
natominit = 6
in the input-file. This is assumed to be the initial value for each sub-run of the G42+-program. If one wants different numbers, one can define different numbers for different walkers, and perform the runs for different numbers of atoms sequentially.

7.21 Set initial temperature

The command 'setTinit'(r.eal) = sets the initial temperature of a simulated annealing run, where r.eal is a real floating point variable in the F-format (like the pressure in the 'setpress' command). Note that for multi-walker runs, all Tinit are set equal to this value. This command is currently rarely used. If one wants different starting temperatures, one can define different temperatures for different walkers, and perform the runs for different starting temperatures sequentially.

7.22 Set volume factor

The variable 'volfactor' determines how much larger the initial cell volume will be than the volume of the single atoms taken together. There are two ways to do this: directly in the input file, and by using the setvolfactor-command.

7.22.1 Method description

The command for setting a new 'volfactor' is command(..) = 'setvolfactor(r.eal)'. Here, the real number r.eal is the new 'volfactor'. The format is F4.2.

7.22.2 Entries of parameters (in input.f)

In the input file, the entry is
volfactor = 3.0d0

7.23 Set moveclass

Several moveclasses can be defined in G42+. The 'setmoveclass'-command allows the switch to a new moveclass, indicated by the value of 'mclflag'. The default value of the moveclass is 'mclflag = 1', explicitly set in the input file. This default value is automatically set at the beginning of simulated annealing, Monte Carlo, multi-quench and threshold runs. In prescribed-path and local/shadow runs, there is an automatic switch from 'mclflag = 1' to 'mclflag = 2': 'mclflag = 1' is for the prescribed path / general exploration, while 'mclflag = 2' is for the local relaxation etc. Finally, if we perform a second minimization, we use a second local moveclass, whose default value is 3.
The use of freely choosing different moveclasses is envisioned for optimizing structures via a sequence of different quench runs, each with a different type of moveclass.

7.23.1 Method description

The command for setting a new moveclass is command(..) = 'setmoveclass(i)'. In order to vary the moveclass in both local and non-local modules independently, in the following, it is assumed that not more than 9 different moveclasses are defined (Thus: $3 \leq$ mclassnum $\leq$ moveclassmax ≤ 9).
i should be a two digit integer between 000 and 999, i = jkl, with the following interpretation:

1. j = 0: The moveclass for local modules used for the second (set of) minimization routines are set to the default value of mclflag_local2: mclflag = 3
2. k = 0: The moveclass for local modules mclflag_local used for the first (set of) minimization routines are set to the default value of mclflag_local: mclflag = 2
3. l = 0: The moveclass for all non-local modules are set to the default value mclflag = 1
4. 1 < j < 9: moveclass for second Local modules are set to value j.

5. 1 < k < 9: moveclass for first local modules are set to value k.

6. 1 < l < 9: moveclass for non-local modules are set to value l.

Thus: i = 0 implies that all modules are set to default value. Note that with 'setmoveclass' we always affect both the local and non-local moveclasses. Thus, we have now three parameters that are active and determine, which moveclass is being employed: j, k and l !

7.23.2 Entries of parameters (in input.f)

The explicit initial setting of the moveclass is given in input.f. The standard initial choice of moveclass is moveclassflag ('mclflag') = 1. Since this is the default value, we also set 'mcldefaultflag' = 0. This means that in all non-local modules, we employ 'mclflag' (= 'mclflag_global') = 1. Remember that for local runs, the default value for 'mclflag' (= 'mclflag_local') = 2, and for a second minimization 'mclflag' (= 'mclflag_local2') = 3, respectively. At the very beginning, we usually set 'mclflag_local', 'mclaflag_local2', and 'mclflag_global' all three equal zero, in order to indicate that we are using the default values.
mcldefaultflag = 0
mclflag = 1
mclflag_global = 0
mclflag_local = 0

7.24 Set abinitinput

The variable 'abinitinputflag' is set by the command 'setabinitinputflag'. It indicates which of several possible ab initio inputs are chosen for calculations. The default value is 'abinitinputflag = 1', set explicitly in the input file.

7.24.1 Method description

The command for setting the 'abinitinputflag' is command(..) = 'setabinitinputflag(i
Here, the integer i constitutes the value of 'abinitinputflag'.

7.24.2 Entries of parameters (in input.f)

The explicit initial setting is in the input as
abinitinputflag = 1

7.25 End of metascript

The command 'end' indicates to G42+ that no further valid commands are coming. Upon encountering this command, G42+ terminates the run in an orderly fashion. Note that you are always required to have this command at the end of your metascript.

7.25.1 Method description

The command for ending the complete run is command(..) = 'end'.
This command has no parameters.

7.25.2 Entries of parameters (in input.f)

This command has no parameters.

7.26 setfixflag

The variable 'fixflag', which describes whether a background structure should be ignored or not, can be changed with the command 'setfixflag(i)' = ...
This command changes the value of fixflag. For this to make sense, one needs to have set 'fixflag' = 1 in input.f for loading the background structure when initializing G42 (see section on background structure for more details). 'fixflag' = 0 means that the background atoms are ignored from now on. If one reactivates 'fixflag' by setting it to 1 again, these atoms 'reappear', however!This can lead to massive overlaps if the rest of the structure has moved in the mean-time, thus, such a re-activation should be used only in special cases, or if one starts a new run and thus needs to have the original background structure be present.

7.27 Setting number of walkers

- 'setwalkcurr': The command 'setwalkcurr'(walk1,walk2) sets the range of the current walker to lie between walk1 and walk2. This is to be used when running a small group of walkers, but one usually employs the standard values 'walk1' = 1 and 'walk2' = 'nwalk'. Keep in mind that when you do some calculations only for a subset of the walkers, and then return to all walkers, you might be mixing explorations that have proceeded in quite some different fashion. Note that 'setwalkcurr' and using the mws- or mwi-type option of module always go together.

- 'seticur': The command 'seticurr(i)' sets the value of the single current walker to 'i'. For single-walker runs, this

value should be between 1 and nwalk (< nwalkmax). This command
should always be used when running a single walker module (or
at the beginning of a sequence of single-walker modules), and
picking the walker for which this should take place. An alternative
commonly used is to use the mws-option of the module of interest
and use the command 'setwalkcurr'(walk1,walk1) to pick the
desired walker (essentially setting 'icurr' = walk1).
Note that after calling any of the multi-walker modules (mws-
or mwi-type), 'icurr' is automatically set to zero; thus one
needs to reset 'icurr' to the desired value afterwards, if
one wants to perform a single walker run.
Furthermore, the only time 'icurr' is to be used with the value
0 is for the initialization routines that generate the 'reference'
configuration when starting G42+.

8 Moveclass

The moveclass is a list of possible moves during a random walk on
the energy landscape. Each move is called a certain fraction of
the time as prescribed by the moveclass. In general, three types
of moves can be employed: Small moves for single walkers, large
moves for single walkers (so-called jumpmoves), and multi-walker
moves that combine information from several walkers to generate
new configurations. In many ways, the small and large moves for
single walkers are qualitatively quite similar, but with the jumpmoves,
we can perform a local run (typically one or two local minimizations)
after executing the moves before its acceptance is decided upon.

The moveclass is the same for all the walkers. If one wants the
walkers to perform runs with different kinds of acceptance probabilities,
one can assign e.g. different temperature programs for each walker
during e.g. a simulated annealing run, even for interacting walkers.
But the length of the runs must be the same for the interacting
multi-walker runs (mwi-modules). Since separate multi-walker runs
(mws-modules) are performed one after the other, the run lengths
can be different in principle - however this would not be very efficient
for a parallel-computer-implementation, of course, where each walker
has its own processor and one wants to avoid idle processors.

Furthermore, we can enhance / decrease the acceptance probability
of a given type of single walker or jump move (at non-zero / non-infinity
temperature), by setting the factor 'moveacceptfact' to a value
different from 1.0. Whenever we compute the acceptance probability,
we multiply the temperature by this factor, i.e., if 'moveacceptfact'
$>$ 1, then this move is more likely to be accepted, and if 'moveacceptfact'

$<$ 1, then the move is less likely to be accepted than for the standard value 'moveacceptfact' = 1.

In order to be more flexible, we are going to introduce several ('mclassnum' $\leq$ 'mclassnummax') moveclasses. Note that 'mclassnum' must be at least 3 (even if we might define them to be the same), and must not be larger than 9. The different types of moveclasses are activated by the command 'setmoveclassflag' (see subsection on setting the moveclass), and all of them need to be defined beforehand in the input.f file, even if one does not want to use e.g. the multi-walker moveclass. Each entry in the moveclass is given as an array, with one of the fields indicating the number of the moveclass, e.g. 'moveTflag(1)' = ...; 'moveTflag(2)' = ..., etc.

Typically, there are two types of entries in the moveclass. First, there are the parameters that define step sizes, both general ones that apply for many moves and specific ones that apply for only one of the moves - in the latter case, these parameters are usually defined as part of the specific move. These parameters can often be adjusted as function of the current temperature, in order to keep the acceptance probability relatively constant by e.g. making the step size smaller at lower temperatures. The second type of entry deals with the specific moves and defines the probability to attempt these moves plus the acceptance probability factor of each move. For each of the three types of moveclasses (standard single, jumpmove and multiwalker-move), the sum over these probabilities has to add to one (= 100 %) separately.

Finally, there are two sets of moves in the single-walker moveclass that are not allowed to be mixed: Those moves that refer to atoms/building units and cells in space group P1 (which is the standard exploration scheme), and those that consist of symmetry adapted moves for space groups that differ from P1. These latter moves are usually used to check whether a particular given structure is a minimum on the energy landscape for fixed symmetries. In that case, only those atom and cell parameters that are still free to be varied for a given space group are allowed to be changed. These are the so-called free-parametermoves and are listed in moves 18 and 19 in the single walker standard moveclass. Note that either moves 18 and 19 are together selected with 100 % probability or the probability of all the other moves together (1 - 17 plus 20 - 25) must add up to 100 %.

8.1 General parameters for standard and jump moves of single walkers

```
At the beginning, there are some lines where we initialize the probabilities
to initial values (all are zero), and all accept factors (all are
one). Do not change these entries.
do i1 = 1,mclassnum
do i2 = 1,mflagmax
initmflag(i1,i2) = 0.0d0
initjumpmflag(i1,i2) = 0.0d0
initmultmflag(i1,i2) = 0.0d0
moveacceptfact(i1,i2) = 1.0d0
enddo
enddo
   Similarly, 'nmovesingle' is the total number of regular single
walker moves. We would only change this value, if a new moveclass
has been added in moveclasssingle.f, else you will get an error
message. Currently, we have 25 single walker standard moves.
nmovesingle = 25
   Finally, we usually do not change the entries for the initial
choice of which moveclass is to be active, and use 'mclflag' = 1.
Since this is the official default value, we also set 'mcldefaultflag'
= 0. This setting of 'mcldefaultflag' means that in all non-local
modules, we employ 'mclflag' (= 'mclflag_global') = 1. Remember
that for local runs, the default value for 'mclflag' (= 'mclflag_local')
= 2. This is the reason we need to list at least two moveclasses
in input.f. Furthermore, at the beginning, we set 'mclflag_local'
and 'mclflag_global' both equal zero, in order to indicate that
we are using the default values.
mcldefaultflag = 0
mclflag = 1
mclflag_global = 0
mclflag_local = 0
```

8.1.1 Movement in two dimensions

```
Usually, both atoms and cell parameters can be changed in three
dimensions. However, this can be restricted by setting the flag
'move2dflag'(mclass): If we set 'move2dflag' = 1, then only the
locations in the x-y-plane can be changed. This is of interest,
if we want to find optimal 2d-structures, or only want the study
the landscape of atoms moving in contact with a surface.
   Usually, no rotations of building units should be attempted in
such a case - an explicit rotation around the z-axis of the general
```

coordinate system is not implemented in moveclass. An exception would be if we had aligned one of the Euler-axes of the molecule with the z-axis of the general coordinate system - in such a case a rotation about that axis would be okay. Rotations of molecules are not forbidden explicitly for two-dimensional explorations, since under certain circumstances a rotation of the building units while keeping the centers of the building units on a given plane might be of interest.

Similarly, keep in mind that exchange moves can include atoms that have different z-values. (if useful, modify exchange moves to refer to only atoms with the z-same value?) If several z-planes are used, one might actually want to move between such planes as part of single atoms displacements. In that case, set the percentage of such attempts to 'jumpzplane' > 0.0d0. (see below)

For jumpmoves, be careful when trying to exchange slices or invert slices of the structure in the 2d-mode. Also, regarding multi-walker moves, one needs to ensure that the length of the cell in z-direction is the same for all walkers before attempting such moves. Also, keep in mind that moves which mix slices of different walkers can switch the z-values of the atoms (all atoms within one plane will move together). Such moves should be avoided in the 2d-mode unless a specific purpose is aimed at.

move2dflag(1) = 0
move2dflag(2) = 0

If we deal with mobile background atoms that are supposed to move inside a 2d-plane, we use 'move2dfixflag'(mclass), analoguosly to 'move2dflag', and 'jumpzplanefix', analogous to 'jumpzplane'.

move2dfixflag(1) = 0
move2dfixflag(2) = 0

8.1.2 Temperature dependence of moves

In order to take into account that at low temperatures most of the larger moves will be rejected and in order to allow a more finely tuned search for the local minima, one can adapt some of the step-sizes to the temperature by setting various flags and parameters.

- The variable 'moveTflag'(mclass) decides on whether the stepsizes are fixed or adaptive
  1. 'moveTflag' = 0 : Moveclass mit fixed value of step-sizes independent of temperature
  2. 'moveTflag' = 1 : Moveclass mit temperature adapted step-sizes.

moveTflag(1) = 1
moveTflag(2) = 1

- The flags 'dcellTflag', 'dTflag', 'frTflag', 'frT2flag' indicate that the move will be adaptive for different temperatures. Note that there is no point in activating 'frTflag' unless 'dcellTflag' or 'dTflag' are active. If these flags are set equal to zero, then the maximal values of the corresponding parameters are employed:
'dmax' is the maximal size (in Angstrom) of the shift of an individual atom, 'dcellmax' is the maximal size (in Angstrom) of the shift in a component of the cell vectors, 'frmax' and 'frT2max' are the maximum degree to which the random number can influence the step-sizes of atoms and cell vectors, respectively.

  They enter into the formulae for the atom shift and the cell shift as follows ('rnumber' is a random number between zero and one):
'd' = 0.5 * 'dT' * (1 + 'frT' * ('rnumber' - 0.5)),
'dcell' = 0.5 * 'dcellT' * (1 + 'frT' * ('rnumber' - 0.5)),
and 'a(i1,i2)' = 'a(i1,i2)' - ('shift' * 'a(i1,i2)')
with 'shift' = ('rnumber' - 'shiftfr2') * 'frT2'.

  'frT2' is used when the cell vectors are changed without changing the atom positions themselves, i.e., a slice of a certain thickness is removed from the cell, but the positions of the atoms in cartesian coordinates are unchanged (of course, the fractional coordinates refering to the cell vectors change!). 'dcell' is used when the cell vectors are changed and the fractional coordinates of the atoms remain unchanged, i.e. the atoms move relative to each other in cartesian coordinates due to the change in cell vector.

  Quite generally, the flags are set equal to zero, then the maximal values are employed (Suggested values for 'dmax', 'dcellmax', 'frmax' and 'fr2max' are: 'dmax' = ??, 'dcellmax' = ?? $<$ 'dmax', 'frmax'= 2.0, 'fr2max' = 0.1, 'frfactmax' = 0.1.

  1. 'dTflag' = 1 gives 'dT' = 'dmax' * $\log_{10}$(T * 10000 + 1) / 'cinf'.
     dTflag(1) = 1
     dTflag(2) = 1
     dmax(1) = 0.05d0
     dmax(2) = 0.05d0
  2. 'dcellTflag = 1' gives 'dcellT' = 'dcellmax' * $\log_{10}$(T * 10000 + 1) / 'cinf', where 'cinf' = $\log_{10}$(10000 * 'Tinf'

```
+ 1) is calculated in the subroutine 'inpcalcwrite'.
dcellTflag(1) = 1
dcellTflag(2) = 1
dcellmax(1) = 0.05d0
dcellmax(2) = 0.05d0
```

3. ```
   'frTflag' = 1 gives 'frT' = 'frmax' * ('fract' * 'dT')
   / (1 + ('fract' * 'dT')**2).
   frTflag(1) = 0
   frTflag(2) = 0
   frmax(1) = 0.1d0
   frmax(2) = 0.1d0
   fract(1) = 1.0d0
   fract(2) = 1.0d0
   ```

4. ```
   'frT2flag' = 1 gives 'frT2' = 'fr2max' * ('fract2' * 'dcellT')
   / (1 + ('fract2' * 'dcellT')**2).
   frT2flag(1) = 0
   frT2flag(2) = 0
   fr2max(1) = 0.15d0
   fr2max(2) = 0.15d0
   fract2(1) = 1.0d0
   fract2(2) = 1.0d0
   ```

5. ```
   The general formula for a cell change is 'a(i1,i2)' = 'a(i1,i2)'
   - ('shift' * 'a(i1,i2)') with 'shift' = ('rnumber' - 'shiftfr2')
   * 'frT2'. Note that if 'shiftfr2' = 0.0, we can only remove
   slices of the cell.
   shiftfr2(1) = 0.50d0
   shiftfr2(2) = 0.50d0
   ```

6. ```
   In moves 14 - 17 and 19, we employ 'frfactTflag' and 'a(...,...)'
   = 'a(...,...)' * 'factcell', where 'factcell' = 1 + 'frfactT'
   * ('rnumber' - 0.5), and 'frfactT' = 'frfactmax' * ('fract2'
   * 'dcellT') / (1 + ('fract2' * 'dcellT')**2).
   frfactTflag(1) = 0
   frfactTflag(2) = 0
   frfactmax(1) = 0.1d0
   frfactmax(2) = 0.1d0
   ```

- ```
  2*pi*'deuler' is the maximal change in an Eulerangle. If we
  make random rotations of only one building unit starting from
  the same reference configuration, it is best to set 'deuler'
  = 1.0d0
  deuler(1) = 0.5d0
  deuler(2) = 0.5d0
  ```
  ```

- 'drs' is the maximal change in the radiusfactor during a step. This is only of interest, if we deal with pseudo-atoms consisting of many electrons where the volume of this electron-group is part of the optimization.
  drs(1) = 0.1d0
  drs(2) = 0.1d0

- 'dtheta21' gives the maximum rotation angle in move 21, 22 and 25 for moveclasssingle. 'dtheta21' = 1.0 corresponds to rotation up to 2pi, analogous to the Euler angle. Usually one uses relatively small rotations, since they can have quite massive effects.
  dtheta21(1) = 0.2d0
  dtheta21(2) = 0.2d0

- dtheta21j gives the maximum rotation angle in the analogous moves for moveclassjump (nos. 14, 15, 18). Usually, one uses a pretty large value like 1.0, in order to allow for a sizeable rotation in the jumpmove (which is usually followed up by a local minimization).
  dtheta21j(1) = 1.0d0
  dtheta21j(2) = 1.0d0

- If we make changes in the magnetization, we set 'dmagmax' as the maximal allowed incremental change.
  dmagmax(1) = 0.5d0
  dmagmax(2) = 0.5d0

8.2 Lists of possible moves for standard single walkers: shifts / rotations of building units, and cell variation

The arrays 'initmflag(...,...)' give the fraction of moves for the different moveclasses (indicated in first field in the array). The sum over all values of 'initmflag(...,...)' must equal 1.0 for each moveclass individually. We need to list as many moveclasses as given by 'mclassnum'. In addition to the probabilities 'initmflag(...,...)', we have to set the 'moveacceptfact(...,...)' for each move and moveclass.

- 'initmflag(...,1)' is the fraction of building-group-moves (real number) of one building unit, including both rotation and translation. If we use 'move2dflag' = 1 for 'n_zlayers' $>$ 1, we can allow a certain percentage of attempts to hop between two z-layers, given by 'jumpzplane' $>$ 0.0 (but $\leq$ 1.0, of course). The value of 'jumpzplane' also applies to the case of purely translational moves in initmflag(2).

```
initmflag(1,1) = 0.00d0
initmflag(2,1) = 0.00d0
jumpzplane(1) = 0.00d0
jumpzplane(2) = 0.00d0
moveacceptfact(1,1) = 1.0d0
moveacceptfact(2,1) = 1.0d0
```

- 'initmflag(...,2)' is the fraction of building-group-moves (real), where only translations are performed for one unit (particularly useful for building units of size one). Check for the value for jumpzplane under initmflag(1).

```
initmflag(1,2) = 0.00d0
initmflag(2,2) = 0.80d0
moveacceptfact(1,2) = 1.0d0
moveacceptfact(2,2) = 1.0d0
```

- 'initmflag(...,3)' is the fraction of building-group-moves (real), where only rotations are executed. 'eulerangleflag' indicates whether every rotation (0), or only a rotation around the z-axis (1), x-axis (2), or y-axis (3) of the molecule in its reference frame is allowed.

```
initmflag(1,3) = 0.00d0
initmflag(2,3) = 0.00d0
moveacceptfact(1,3) = 1.0d0
moveacceptfact(2,3) = 1.0d0
eulerangleflag(1) = 0
eulerangleflag(2) = 0
```

- 'initmflag(...,4)' is the fraction of moves that exchange the position of two atoms (the second one chosen randomly). Building units are excluded in this case, if their size is larger than 1. Note that for jumpmoves, such molecule exchanges are allowed since the large overlap usually generated can be reduced during a follow-up local minimization.
  In order to be able to do some 'kinetic MonteCarlo' type of simulations, we add an option that restricts the exchange to atoms in the neighborhood. This option is active, if 'exchdistflag(mclflag)' is equal to 1. The radius within which the target atoms is allowed to be located is 'exchangedist(mclflag)'. 'exchpairattempts' gives the number of attempts to find an atom to exchange / to find an atom to serve as partner in the exchange.

```
initmflag(1,4) = 0.00d0
initmflag(2,4) = 0.00d0
moveacceptfact(1,4) = 1.0d0
moveacceptfact(2,4) = 1.0d0
```

```
exchdistflag(1) = 0
exchdistflag(2) = 0
exchangedist(1) = 1.0d3
exchangedist(2) = 1.0d3
exchpairattempts(1) = 1000
exchpairattempts(2) = 1000
```

- 'initmflag(...,5)': Fraction of moves that change the cell vectors, while changing the positions of the building units in cartesian coordinates at the same time, but where the fractional coordinates of the building units are unchanged. Indicate if you also restrict the volume range (see below).

```
initmflag(1,5) = 0.00d0
initmflag(2,5) = 0.10d0
moveacceptfact(1,5) = 1.0d0
moveacceptfact(2,5) = 1.0d0
```

- 'initmflag(...,6)': Fraction of moves that change the cell vectors, while not changing the positions of the building units (in cartesian coordinates) at the same time, i.e. the fractional coordinates of the building units are changed. This corresponds to taking a slice out of the unit cell. Any atom that are in the slice that is taken out are moved into the next 'periodically repeated cell' and then shifted back by a lattice translation into the actual simulation cell. Indicate if you also restrict the volume range (see below).

```
initmflag(1,6) = 0.00d0
initmflag(2,6) = 0.05d0
moveacceptfact(1,6) = 1.0d0
moveacceptfact(2,6) = 1.0d0
```

- initmflag(...,7): Fraction of moves to displace the origin of the coordinate system, i.e. a shift of the cell translation lattice with respect to the Cartesian arrangement of the atoms. (perhaps combine it with initmflag6 in the future?!) Used to address cut-offs in the potential. Usually, the energy change is essentially zero for this move.

```
initmflag(1,7) = 0.00d0
initmflag(2,7) = 0.05d0
moveacceptfact(1,7) = 1.0d0
moveacceptfact(2,7) = 1.0d0
```

- 'initmflag(...,8)': Generation of a pair of charges via transfer of one unit of charge from one atom to another (only for building units of size 1).

```
initmflag(1,8) = 0.00d0
initmflag(2,8) = 0.00d0
moveacceptfact(1,8) = 1.0d0
moveacceptfact(2,8) = 1.0d0
```

- 'initmflag(...,9)': Removing an atom, but leaving its charge; only for building units of size 1 (Currently not active).
```
initmflag(1,9) = 0.00d0
initmflag(2,9) = 0.00d0
moveacceptfact(1,9) = 1.0d0
moveacceptfact(2,9) = 1.0d0
```

- 'initmflag(...,10)': Adding a neutral atom; only for building units of size 1 (Currently not active). 'simax' and 'tol1' are parameters needed for such a move. 'simax' gives the number of steps in the i direction while searching for holes in the structure to place the atom. 'sleni' = 1.0/'simax' is the size of these steps in units of the current cell. If the dimension 'dimens' = 2, set 's3max' = 1. 'tol1' is the required tolerance for assuming that an atom is far enough away from the place of the hole during the subroutine holesearch. One should probably pick twice the maximal radius available among the participating atoms.
```
initmflag(1,10) = 0.00d0
initmflag(2,10) = 0.00d0
moveacceptfact(1,10) = 1.0d0
moveacceptfact(2,10) = 1.0d0
s1max = 10
s2max = 10
s3max = 10
tol1 = 3.63d0
```

- 'initmflag(...,11)': Exchange of atom of type a for one of type b; only for building units of size 1 (Currently not active).
```
initmflag(1,11) = 0.00d0
initmflag(2,11) = 0.00d0
moveacceptfact(1,11) = 1.0d0
moveacceptfact(2,11) = 1.0d0
```

- 'initflag(...,12)': Addition of a slice of cell plus an atom (like in 'initmflag(...,6)'); only for building units of size 1 (Currently not active).
```
initmflag(1,12) = 0.00d0
initmflag(2,12) = 0.00d0
```

```
moveacceptfact(1,12) = 1.0d0
moveacceptfact(2,12) = 1.0d0
```

- 'initmflag(...,13)': Change of the radiusfactor of an ion, only for building units of size 1.
```
initmflag(1,13) = 0.00d0
initmflag(2,13) = 0.00d0
moveacceptfact(1,13) = 1.0d0
moveacceptfact(2,13) = 1.0d0
```

The following four moves are restricted cell moves, i.e. we try to keep certain (usually symmetry-related) aspects of the cell fixed. Unless explicitly necessary, these moves should not be mixed with the arbitrary volume changes listed above.

A particular special case is the one, where the symmetry of the configuration is given, and thus a small set of (free) cell and atom parameters needs to be varied, which affect many atoms, while some atoms can be treated as completely fixed. These moves can be mixed with restricted cell moves, which respect the symmetry of the structure. The moves 14 - 17 allow us to focus on special aspects of the cell parameters alone of the symmetric configuration, while moves 18 - 19 are for considering both all atom and all cell symmetry adapted parameters. In this specific case, the earlier warning (of never mixing moves 18 and 19 with any of the other moves in the single walker standard moveclass) can be ignored.

- 'initmflag(...,14)': Rescaling of the whole cell, while keeping the relative positions of the atoms fixed (analogous to 'initmflag(...,5)', where only individual cell vectors were rescaled).
```
initmflag(1,14) = 0.0d0
initmflag(2,14) = 0.0d0
moveacceptfact(1,14) = 1.0d0
moveacceptfact(2,14) = 1.0d0
```

- 'initmflag(...,15)': Rescaling of cell vectors, while keeping the angles between them constant. This would preferably be used for the 'idealized' cell input: (a,0,0),(b1,b2,0),(c1,c2,c3). For 'initmflag(...,15)', we also need to set the flag 'factcellflag(...)', which specifies the type of scaling.

  1. 'factcellflag(...)' = 1: scale a, while keeping b/a and c/a constant
  2. 'factcellflag(...)' = 2: scale a or c, while keeping b/a constant

3. 'factcellflag(...)' = 3: let all cell length vary keeping only the angles constant
4. 'factcellflag(...)' = 4: a,b,alpha=beta=90^o, are kept constant, while c1 or c3 are varied. Thus, for a monoclinic system we can vary specifically the angle gamma.
5. 'factcellflag(...)' = 5: Full variation allowed in the monoclinic system.
6. 'factcellflag(...)' = 6: Rescale only the third cell vector; this is particularly useful for changing the cell when using a fixed slab in the x-y plane.

Both for 'factcellflag(...)' = 4 and 'factcellflag(...)' = 5, do we require that the description of the cell is such that b1 = c2 = 0. Do remember to set 'initmflag(...,15) to zero unless you want to restrict the cell moves. If we want only changes in the shape of cell in the x-y-plane, one would probably prefer to use the 'move2dflag' (however, keep in mind that this would also restrict the atom moves to only shifts parallel to the x-y-plane).

```
initmflag(1,15) = 0.0d0
initmflag(2,15) = 0.0d0
factcellflag(1) = 0
factcellflag(2) = 0
moveacceptfact(1,15) = 1.0d0
moveacceptfact(2,15) = 1.0d0
```

- 'initmflag(...,16)': Rescaling of cell vectors, while keeping the total volume constant. This would preferably be used for the 'idealized' cell input: (a,0,0),(b1,b2,0),(c1,c2,c3). For 'initmflag(...,16)', we also need to set the flag 'vol0flag(...)', which tells us, whether we are to scale a or c, while keeping b/a constant ('vol0flag(...)' = 2), or let all vary keeping only the angles constant ('vol0flag(...)' = 3). These are the only choices available for such constant volume moves. Note that for constant volume moves, it makes most sense to have prescribes an initial cell volume (unless we read in a starting configuration), since the density of the automatically generated simulation cell is usually much too low (or too large, for zeolites). Do remember to set 'vol0flag(...)' to zero unless you want to use the fixed volume option.

```
initmflag(1,16) = 0.0d0
initmflag(2,16) = 0.0d0
vol0flag(1) = 0
vol0flag(2) = 0
```

```
moveacceptfact(1,16) = 1.0d0
moveacceptfact(2,16) = 1.0d0
```

- 'initmflag(...,17)': Rescaling of cell vectors, while keeping the volume between 'volrangetop' and 'volrangebottom'. This would preferably be used for the 'idealized' cell input: (a,0,0),(b1,b2,0),(c1, For 'initmflag(...,17), we also need to set the flag 'volrangeflag(...)', which tells us, how we are supposed to scale the cell parameters.

  1. 'volrangeflag(...)' = 1: scale a, while keeping b/a and c/a constant
  2. 'volrangeflag(...)' = 2: scale a or c, while keeping b/a constant
  3. 'volrangeflag(...)' = 3: let all vary keeping only the angles constant
  4. 'volrangeflag(...)' = 4: a,b,alpha=beta=90o, are kept constant, while c1 or c3 are varied. Thus, for a monoclinic system we can vary specifically the angle gamma.
  5. 'volrangeflag(...)' = 5: Full variation in the monoclinic system

  Both for 'volrangeflag(...)' = 4 and 'volrangeflag(...)' = 5, do we require b1 = c2 = 0. Do remember to set the 'volrangeflag(...)' to zero unless you want to restrict the cell moves.

```
initmflag(1,17) = 0.0d0
initmflag(2,17) = 0.0d0
volrangeflag(1) = 0
volrangeflag(2) = 0
moveacceptfact(1,17) = 1.0d0
moveacceptfact(2,17) = 1.0d0
```

- Volume range restriction during cell moves: We need a flag 'volcheckflag(...)' that controls whether there are explicit restrictions to the volume range that the walker is allowed to explore. Note that this volume range restriction is not only active for moves 14 - 17 but also for the completely free variation of the cell volume under moves 5 and 6.

  1. 'volcheckflag(...)' = 1: only 'volrangebottom' needs to be considered
  2. 'volcheckflag(...)' = 2: both a lower and an upper bound is required
  3. 'volcheckflag(...)' = 0: no restriction on the overall volume given (but do check the general restrictions regarding

```
        the volume of the cell compared to the total volume of
        the atoms). This is usually the option for the moves 5
        and 6.

    'volrangebottom' and 'volrangetop' are given in Ang**3. If
    the optimization is to stay within a volume range, it is necessary
    to prescribe an initial volume that lies within the range,
    unless a specific configuration is being read in at the outset
    (whose volume should be within the prescribed range, of course).
    If only a minimum volume is prescribed, we can usually start
    with a large initial volume as one does in a standard exploration.
    Do remember to set the flags 'volcheckflag(...)' and 'volrangeflag(...)'
    (see above) to zero unless you want to use the volume range
    limitation.
    volcheckflag(1) = 0
    volcheckflag(2) = 0
    volrangebottom(1) = 1000
    volrangebottom(2) = 100
    volrangetop(1) = 1600
    volrangetop(2) = 200
```

8.2.1 Free parameter moves (moves adapted to space group symmetry)

```
If we want to use free parameters of a structure with given symmetry,
then we set 'freeparamflag' = 1, and us this set of moves for changes
in atom positions. Remember that the 'free parameters' refer to
all those atom and cell variables that can be changed while still
preserving the symmetry of the structure - all the other variables
are fixed by symmetry once the free parameters are set. Typically,
such explorations only take place when we load in a structure that
has a certain symmetry given by its space group, and we only want
to vary the free parameters that leave the symmetry of the structure
unchanged. The number of degrees of freedom, i.e. the number of
free parameters of the atoms, 'nfapcur' plus the cell lengths and
angles, 'nLfap' and 'nWfap', respectively, are initialized here.
Furthermore, we initialize all the free-parameter arrays (to zero).
Note that these are overwritten when loading in the starting configuration
- this configuration must already contain all the information needed
to identify the free parameters in the system. Usually, one employs
the GASP-command in KPLOT to find these free parameters.
freeparamflag(1) = 1
freeparamflag(2) = 0
nfapcur = 0
```

```
nLfap = 3
nWfap = 3 Make sure that moves 18 and 19 are not mixed with the
moves 1 - 17 and 20 - 25, i.e., the sum of 'initflag(...,18)' and
'initflag(...,19)' must be equal to 1 and the sum of 'initflag(...,1)'
to 'initflag(...,17)' plus 'initflag(...,20)' to 'initflag(...,25)'
must be equal to zero, or the other way around (for all moveclasses
defined). The exception can be mixing some variants of the moves
14 - 17 that are compatible with the symmetry of the structure,
with the free parameters moves 18 and 19.
```

- ```
  'initflag(...,18)' makes changes in freely variable atom parameters.
  Recall that with free parameter runs, we cannot have building
  units of size larger than 1, due to the symmetry restrictions
  (moving a molecule while keeping all the symmetries of the
  structure is a rather complex operation in general).
  initmflag(1,18) = 0.40d0
  initmflag(2,18) = 0.00d0
  moveacceptfact(1,18) = 1.0d0
  moveacceptfact(2,18) = 1.0d0
  ```

- ```
  'initflag(...,19)' makes changes in cell lengths/angles. Note
  that the starting configurations might correspond to rather
  strangely shaped unit cells that violate restrictions on what
  kind of unit cells are allowed during global exploration runs
  (see configuration section). If necessary, increase these limits.
  initmflag(1,19) = 0.60d0
  initmflag(2,19) = 0.00d0
  moveacceptfact(1,19) = 1.0d0
  moveacceptfact(2,19) = 1.0d0
  ```

8.3 Lists of possible moves for standard single walkers: flexible building units, magnetization, background structure

8.3.1 Changes of flexible building units

```
For large building units, it no longer makes sense to treat them
as rigid, and one wants to introduce flexibility via 'flexbgflag'
instead. However, this requires special types of moves that allow
us to vary not only the position and orientation of the building
unit but also its internal structure, i.e. the internal position
vectors 'uref' that describe the positions of the atoms in the molecule
with respect to some reference point (usually some atom in the center
of the molecule or the center of mass of the initial molecule).
The following moves deal with such building units.
   (Note that in the future, we might want to be more restrictive
with the choice of atoms that are allowed to be involved in a defining
```

role for these large moves. Especially, we might want to exclude
terminal (hydrogen) atoms. This would involve changes in the moveclass-files,
and perhaps also some restrictions in the computation of free triplets
etc. and axis-atoms in inpclcwr.f in order to have smaller sets
of moves and fewer uninteresting moves. But at the moment we are
very general in our moves.)

- 'initmflag(...,20)' varies positions of individual atoms, pairs,
  triplets and quartets according to probability 'move20flag'(mclflag,1/2/3/4),
  respectively. Make sure that the 'move20flag'-entries add up
  to 1.0 for each moveclass. Keep in mind the number of atoms
  vs. number of pairs vs. number of triplets vs. number of quartets
  when selecting the 'move20flag'-entries. The pairs/triplets/quartets
  are shifted by the same amount, changing the 'uref'(...) for
  the randomly chosen building unit.
  initmflag(1,20) = 0.00d0
  initmflag(2,20) = 0.00d0
  move20flag(1,1) = 0.25d0
  move20flag(1,2) = 0.25d0
  move20flag(1,3) = 0.25d0
  move20flag(1,4) = 0.25d0
  move20flag(2,1) = 1.00d0
  move20flag(2,2) = 0.00d0
  move20flag(2,3) = 0.00d0
  move20flag(2,4) = 0.00d0
  moveacceptfact(1,20) = 1.0d0
  moveacceptfact(2,20) = 1.0d0

- 'initmflag(...,21)' changes the building unit by rotating a
  piece of the structure that can be disconnected by breaking
  the free atom pair AB (inside a sequence XiABYi; A and B can
  be branch nodes) around either the axis XiA or BYi, where we
  move the atoms B plus the rest of the molecule on the B-side
  of the breakable pair AB (or atoms A and the rest on the A-side
  of the pair AB). Again, the change is effected by changing
  the 'uref'(...)-values of the atoms in the building unit.
  initmflag(1,21) = 0.00d0
  initmflag(2,21) = 0.00d0
  moveacceptfact(1,21) = 1.0d0
  moveacceptfact(2,21) = 1.0d0

- 'initmflag(...,22)' changes the building unit by changing distances/angles/dihe
  that separate the building unit into two unconnected pieces.
  Again, the change is effected by changing the 'uref'(...)-values
  of the atoms in the building unit. The probability of attempts

```
to change distances / angles / dihedrals is given by 'move22flag'(mclflag,1/2/3
respectively. Make sure that these values add up to 1 for each
moveclass.
initmflag(1,22) = 0.00d0
initmflag(2,22) = 0.00d0
move22flag(1,1) = 0.00d0
move22flag(1,2) = 0.50d0
move22flag(1,3) = 0.50d0
move22flag(2,1) = 0.00d0
move22flag(2,2) = 0.50d0
move22flag(2,3) = 0.50d0
moveacceptfact(1,22) = 1.0d0
moveacceptfact(2,22) = 1.0d0
```

- 'initmflag(...,25)' deals with a move for the flexible building unit, where we select two atoms inside the building unit, and then take one set of atoms that lie between these two atoms, and rotate this set around the axis connecting the two atoms.

```
initmflag(1,25) = 0.00d0
initmflag(2,25) = 0.00d0
moveacceptfact(1,25) = 1.0d0
moveacceptfact(2,25) = 1.0d0
```

8.3.2 Changes in magnetization

Currently varying magnetization makes only sense with QE-calculations. Also, set 'magnetflag' = 3.

- 'initmflag(...,23)' changes individual settings of the magnets ('magnetatom' - do not confuse with the magnetization, which is the output of the QE-calculation).

```
initmflag(1,23) = 0.00d0
initmflag(2,23) = 0.00d0
moveacceptfact(1,23) = 1.0d0
moveacceptfact(2,23) = 1.0d0
```

8.3.3 Changing the background structure

The common use of a background structure is as an unmovable set of background atoms that provide the environment of the mobile atoms and building units. However, we might want to consider small relaxations of the background structure as part of the moveclass, especially for the local minimizations of already pre-optimized structures. In that case, some moves of the atoms and building units that constitute the background structure might be useful as part of a local stochastic

random walk (see also section background structure for more information about background structures and how to keep them fixed.)

- 'initmflag(...,24)' is the fraction of building-group-moves in the backgroundstructure (real atoms, not idealized perfect slab), including both rotation and translation. These are analogous to the moves 1 - 3 for normal mobile atoms - the shape of building units that are part of the background structure cannot be changes with this move (molecules are assumed to be rigid). Note that for this move to be active, we need to have set 'fixflag' = 3, else one just wastes these moves.

  If we use 'move2dflagfix' = 1 for 'n_zlayers' $>$ 1, we can allow a certain percentage of attempts to hop between two z-layers, given by 'jumpzplanefix' $>$ 0.0 (but $\leq$ 1.0, of course). Usually, this kind of move will not be used for a background structure since this structure is supposed to be already pretty well optimized, and we only look for small relaxations of the background in response to the atoms, molecules and clusters on the surface or inside the background structure. Thus, we also do not permit any changes in cell parameters. If such large changes are desired, one should enter the 'background structure' as a list of regular atoms/building units with specified positions and specific choices of 'fixbg' as function of moveclass, and let this structure relax on its own (with or without other molecules) as far as the cell parameters are concerned, and then use the output of this calculation as new background structure.
  initmflag(1,24) = 0.00d0
  initmflag(2,24) = 0.00d0
  jumpzplanefix(1) = 0.00d0
  jumpzplanefix(2) = 0.00d0
  moveacceptfact(1,24) = 1.0d0
  moveacceptfact(2,24) = 1.0d0

8.4 Large moves (jumpmoves) for single walkers

8.4.1 General aspects

An important option for many modules is the ability to perform large moves on the landscape, often followed by a local minimization of configuration. Sometimes, such jumping on the landscape is given the general name 'basin-hopping', but this term also refers to a particular implementation of such explorations for the purpose of global optimization. Here, the jumpmoves are moves that are performed by single walkers. Many of the jumpmoves are large versions of the standard single walker moves (large displacements of atoms and changes

of cell vectors), while others involve many atoms or building units being shifted and rotated at the same time.

If jump moves are used outside of the salocrun-module (where local runs, especially minimizations, take place by definition), then one frequently wants to perform a local minimization, similar to the one during a multi-walker move (see below). The length and type of the minimization / local run are defined in the poolmove section (see below). Note that the index counting the acceptance factors equals (30 + number of jumpmove).

- 'fractjump(mclflag)' indicates the fraction of single walker moves that are very large jump moves. This fraction depends on the moveclass. Since moveclass 2 is nearly always used for the local minimization as default, fractjump(2) should usually be set equal to 0. (As a precaution, wherever a local minimization takes place automatically, only calls to moveclasssingle are performed by default during such a local run.)
  fractjump(1) = 0.1d0
  fractjump(2) = 0.0d0

- 'nmovejump' is the number of types of (active) single walker jump moves. Do not change this number unless moveclassjump.f has been changed.
  nmovejump = 18

8.4.2 General parameters specific to many single walker jump moves

'initjumpmflag(...,...)' indicates percentages of large moves involving many atoms and cell parameters. To set the size of such moves, the following parameters are needed:

- 'djumpfact': factor to multiply the value of the stepsize of atom shifts 'd' from standard single walker moves

- 'numexchjump': number of exchanges of single atoms that take place during a multi-exchange move. Note that the value for 'exchpairattempts(mclflag)' (how often the walker tries to finds two atoms or building units that can be be exchanged) is defined as part of the single-walker moveclass.

- 'dcelljumpfact': factor to multiply the stepsize of cell changes 'dcell' from standard single walker moves

- 'shiftjumpfact': factor to multiply the shift parameter 'shift' from standard single walker moves

- 'nshufflebins': number of slices to partition a cell in, before shuffling
- 'ntupleslice': number of times one slice of the cell is multiplied to generate a new cell. (E.g., if 'ntupleslice' = 3, then one third of the cell (A) is kept and the other two thirds (B) are removed and replaced by two slices A.) The corresponding move (jumpmove 18) is not yet active.
- 'dbox' is the size of the box (+- 'dbox' in the x, y and z direction measured in relative units of the cell vector) around a randomly picked atom, where the atoms inside the box are moved ('dbox' must be smaller than 0.5, else it would include all atoms in the cell. For example, the atoms in the box can be randomly interchanged. If we only shift the atoms that are in the box, then they are moved by (rnumber- 0.5)*'dboxfract' * 'dbox' in all three direction x,y,z.

```
djumpfact = 5.0d0
dcelljumpfact = 5.0d0
shiftjumpfact = 5.0d0
numexchjump = 10
nshufflebins = 4
ntupleslice = 2
dbox = 0.2
dboxfract = 0.5
```

8.4.3 List of jumpmoves for atoms and rigid building units

- 'initjumpmflag(...,1)': Shift and rotation of all atoms
  ```
  initjumpmflag(1,1) = 0.0d0
  initjumpmflag(2,1) = 0.0d0
  moveacceptfact(1,31) = 1.0d0
  moveacceptfact(2,31) = 1.0d0
  ```
- 'initjumpmflag(...,2)': Only shift of all atoms
  ```
  initjumpmflag(1,2) = 0.2d0
  initjumpmflag(2,2) = 0.2d0
  moveacceptfact(1,32) = 1.0d0
  moveacceptfact(2,32) = 1.0d0
  ```
- 'initjumpmflag(...,3)': Only rotation of all atoms
  ```
  initjumpmflag(1,3) = 0.0d0
  initjumpmflag(2,3) = 0.0d0
  moveacceptfact(1,33) = 1.0d0
  moveacceptfact(2,33) = 1.0d0
  ```

- initjumpmflag(...,4): Multiple exchange among all atoms ('numexchjump' is the number of valid exchanges that take place)
  initjumpmflag(1,4) = 0.1d0
  initjumpmflag(2,4) = 0.1d0
  moveacceptfact(1,34) = 1.0d0
  moveacceptfact(2,34) = 1.0d0

- 'initjumpmflag(...,5)': Change of all cell parameters with co-moving atoms (analogue to standard move 5)
  initjumpmflag(1,5) = 0.1d0
  initjumpmflag(2,5) = 0.1d0
  moveacceptfact(1,35) = 1.0d0
  moveacceptfact(2,35) = 1.0d0

- 'initjumpmflag(...,6)': Change of all cell parameters with fixed atoms (i.e. fixed cartesian coordinates analogue to standard move 6)
  initjumpmflag(1,6) = 0.1d0
  initjumpmflag(2,6) = 0.1d0
  moveacceptfact(1,36) = 1.0d0
  moveacceptfact(2,36) = 1.0d0

- 'initjumpmflag(...,7)': Shift of all atoms and change of all cell parameters with co-moving atoms
  initjumpmflag(1,7) = 0.2d0
  initjumpmflag(2,7) = 0.2d0
  moveacceptfact(1,37) = 1.0d0
  moveacceptfact(2,37) = 1.0d0

- 'initjumpmflag(...,8)': Exchange of two cell slices
  initjumpmflag(1,8) = 0.0d0
  initjumpmflag(2,8) = 0.0d0
  moveacceptfact(1,38) = 1.0d0
  moveacceptfact(2,38) = 1.0d0

- 'initjumpmflag(...,9): Inversion of a cell slice
  initjumpmflag(1,9) = 0.0d0
  initjumpmflag(2,9) = 0.0d0
  moveacceptfact(1,39) = 1.0d0
  moveacceptfact(2,39) = 1.0d0

- 'initjumpmflag(...,10)': Random exchange of atoms within a box around a randomly selected atom
  initjumpmflag(1,10) = 0.0d0
  initjumpmflag(2,10) = 0.0d0

```
moveacceptfact(1,40) = 1.0d0
moveacceptfact(2,40) = 1.0d0
```

- 'initjumpmflag(...,11)': Random shift of atoms within a box around a randomly selected atom
```
initjumpmflag(1,11) = 0.3d0
initjumpmflag(2,11) = 0.3d0
moveacceptfact(1,41) = 1.0d0
moveacceptfact(2,41) = 1.0d0
```

- 'initjumpmflag(...,12)': n-tupling of a cell. This is not yet active.
```
initjumpmflag(1,12) = 0.0d0
initjumpmflag(2,12) = 0.0d0
moveacceptfact(1,42) = 1.0d0
moveacceptfact(2,42) = 1.0d0
```

8.4.4 List of jumpmoves for flexible building units

Similar as for the standard single walker moveclass, for large building units, one wants to introduce flexibility via the flag 'flexbgflag'. The following jump moves deal with such building units, where again the coordinates 'uref' of the atoms in the building unit with respect to the reference point are changed.

- 'initjumpmflag(...,13)' varies positions of many individual atoms, pairs, triplets and quartets according to probability 'movej20flag'(mclflag,1/2/3/4), respectively, in analogy to move 20 of the standard single walker moves. Make sure that the 'movej20flag'-entries add up to 1.0 for each moveclass.

  For the jumpmove, we select several atoms/pairs/etc. when making the configuration change, not only one as for the standard single walker moves. We can choose explicitly the fraction of atoms etc. by setting 'bgfractionjump' to a number larger than zero. The computation of the number of atoms proceeds via 'number of atoms selected' = 'number of atoms in molecule'/'bgfractionjump'

  1. 'bgfractionjump' = 1: All atoms in the building unit
  2. 'bgfractionjump' = 2: Half of the atoms (chosen at random)
  3. 'bgfractionjump' = 3: One third of the atoms (chosen at random)
  4. ...
  5. 'bgfractionjump' = 0: some fraction between one half and one fifth is chosen at random.

bgfractionjump = 0

All atoms belonging to the same pair/triplet/quartet are shifted by the same amount, changing uref(...) for the randomly chosen building unit. Note that each randomly chosen atom / pair etc. gets its own shift. Also, some atoms can be moves several times depending on whether they are chosen several times (or partake in more than one of the pairs/triplets/quartets that are chosen.

```
initjumpmflag(1,13) = 0.00d0
initjumpmflag(2,13) = 0.00d0
movej20flag(1,1) = 1.00d0
movej20flag(1,2) = 0.00d0
movej20flag(1,3) = 0.00d0
movej20flag(1,4) = 0.00d0
movej20flag(2,1) = 1.00d0
movej20flag(2,2) = 0.00d0
movej20flag(2,3) = 0.00d0
movej20flag(2,4) = 0.00d0
moveacceptfact(1,43) = 1.0d0
moveacceptfact(2,43) = 1.0d0
```

- 'initjumpmflag(...,14)' is analogous to move 21 of the standard single walker moves. It changes the building unit by rotating a piece of the structure that can be disconnected by breaking the free atom pair AB (inside a sequence XiABYi; A and B can be branch nodes) around either the axis XiA or BYi, where we move the atoms B plus the rest of the molecule on the B-side of the breakable pair AB (or atoms A and the rest on the A-side of the pair AB). Note that the only difference to the standard move is that one usually would use larger values for 'dtheta21j' than for 'dtheta21'. (In principle, we could perform several such rotations for a large molecule at the same time, but this option has not been implemented.)

  Again, the change is effected by changing the 'uref'(...)-values of the atoms in the building unit.

  ```
  initjumpmflag(1,14) = 0.00d0
  initjumpmflag(2,14) = 0.00d0
  moveacceptfact(1,44) = 1.0d0
  moveacceptfact(2,44) = 1.0d0
  ```

- 'initjumpmflag(...,15)' is the analogue to move 22 of the standard single walker moves. It changes the building unit by changing distances/angles/dihedrals that separate the building unit into two unconnected pieces. Again, the change is effected by changing the 'uref'(...)-values of the atoms in the building

unit. The probability of attempts to change distances / angles
/ dihedrals is given by 'movej22flag'(mclflag,1/2/3), respectively.
These values should add up to 1. Again, the only difference
to the standard moves is the larger value for 'dtheta21j'.
(In principle, we could perform several rotations for a large
molecule at the same time, but this option is not implemented.)
initjumpmflag(1,15) = 0.00d0
initjumpmflag(2,15) = 0.00d0
movej22flag(1,1) = 0.50d0
movej22flag(1,2) = 0.50d0
movej22flag(1,3) = 0.00d0
movej22flag(2,1) = 0.50d0
movej22flag(2,2) = 0.50d0
movej22flag(2,3) = 0.00d0
moveacceptfact(1,45) = 1.0d0
moveacceptfact(2,45) = 1.0d0

- 'initjumpmflag(...,16)' exchanges several building units of
size larger than one, but does not rotate them at the same
time. The maximum number of exchanges is given by 'numexchjumplarge'.
The number of attempts to find a pair of units that can be
exchanged is given by 'exchpairattempts' as in the standard
moves. Note also, that if you have activated 'exchdistflag'
(see move 4 among the standard single walker moves), then you
will only allow the exchange of building units where the first
atoms listed in the building units have a distance smaller
than 'exchangedist'. The reason for this choice is that in
many descriptions of the building unit, the first atom on the
list is either the reference atom or the atom closest to the
center of mass.
initjumpmflag(1,16) = 0.00d0
initjumpmflag(2,16) = 0.00d0
moveacceptfact(1,46) = 1.0d0
moveacceptfact(2,46) = 1.0d0
numexchjumplarge = 1

- 'initjumpmflag(...,17)' aligns several building units by setting
their first two Euler angles equal, and adds a specific angle
(i1 = 1,2,3,4,5,6,7 corresponds to an angle of 0,30,45,60,90,120,180,
respectively) to the third Euler angle of the second building
unit after setting them first equal. The number of building
units that can be aligned is given by 'numalignjumplarge'.
The percentage of attempts for a particular angle are given
by 'alignangle'(i1,mclflag). Make sure that the percentages

```
add up to 1 for each moveclass. Note that only building units
with non-negative 'fixbg' (= 0 or positive number) for the
moveclass can participate in an alignment. The number of attempts
to find such a pair of building units is given by the variable
'exchpairattempts' set in the standard single walker moveclass.
initjumpmflag(1,17) = 0.00d0
initjumpmflag(2,17) = 0.00d0
moveacceptfact(1,47) = 1.0d0
moveacceptfact(2,47) = 1.0d0
numalignjumplarge = 1
alignangle(1,1) = 1.0d0
alignangle(1,2) = 1.0d0
alignangle(2,1) = 0.0d0
alignangle(2,2) = 0.0d0
alignangle(3,1) = 0.0d0
alignangle(3,2) = 0.0d0
alignangle(4,1) = 0.0d0
alignangle(4,2) = 0.0d0
alignangle(5,1) = 0.0d0
alignangle(5,2) = 0.0d0
alignangle(6,1) = 0.0d0
alignangle(6,2) = 0.0d0
alignangle(7,1) = 0.0d0
alignangle(7,2) = 0.0d0
```

- ```
  'initjumpmflag(...,18)' generates a partial rotation move for
  the flexible building unit, in analogy to move 25 of the standard
  single walker moves: we select two atoms inside the building
  unit, and then take one set of atoms that lie between these
  two atoms, and rotate this set around the axis connecting the
  two atoms. The size of the move is controlled by 'dtheta21j'.
  initjumpmflag(1,18) = 0.00d0
  initjumpmflag(2,18) = 0.00d0
  moveacceptfact(1,48) = 1.0d0
  moveacceptfact(2,48) = 1.0d0
  ```

8.5 Multi-walker (interacting) moves

```
For interacting-multi-walker modules, we can generate new configurations
by mixing information taken from two (or more) different walkers.
These so-called cross-over moves are the central element of genetic
(or evolutionary) algorithms for the global optimization of complex
energy landscapes. However, one should note that such explorations
do not produce much information regarding the barrier structure
```
```

of a landscape. Quite generally, one prefers to perform for all the newly generated structures a local minimization since the mixed structures (similar to the configurations generated by jumpmoves) are usually very high in energy and thus one would reject most of them off-hand during e.g. a simulated annealing run (unless one uses very high temperatures, but then one performs essentially an unrestricted random walk, which is not very suitable for optimization purposes.) Once a new set of walkers has been generated, we then use an ensemble acceptance criterion, in order to select from among the old and new walkers the 'best' set of 'nwalk' walkers with which we continue our exploration via further jumpmoves, multi-walker moves or standard single walker moves. What is called 'best' can either be ranking by energy, or ranking by Boltzmann-probability, i.e. by Boltzmann factor multiplied by a random number drawn according to the Boltzmann distribution.

8.5.1 Parameters regarding number of walkers

- 'nwalkdum' is the number of new walkers that can be generated during a mixing step. It should be similar to the total number of walkers 'nwalk'; usually one set 'nwalkdum' = 'nwalk'. Make sure that (nwalk + nwalkdum + 2) is smaller than nwalkmax, in order to have some play in the number of walkers.
  nwalkdum = 0

- For multi-walker-separate runs (i.e. independent walkers) , one can use the 'setwalkcurr' command to define a subset of walkers from 'walk1' to 'walk2' that are being run through. Note that 'walk1' and 'walk2' are automatically being reset to 1 and 'nwalk', respectively, when 'loadnewcfg' or 'gennewcfg' have been active. This holds even if 'countinitdata1' and 'countinitdata2' in these commands are unequal to 1 and 'nwalk', respectively, or if one uses the 'setwalkcurr' command to restrict the subset of active walkers. Similarly, we always set 'walk1' and 'walk2' equal to 1 and 'nwalk' when calling a mwi-type subroutine. The reason for this is that for multi-walker moves the new walkers are generated and then accepted or rejected via an walker ensemble elimination move, and these generation and acceptance routines assume implicitly that the current ensemble contains 'nwalk' walkers.

  Thus one should be very careful when loading/generating only a subset of the total of 'nwalk' configurations before starting a mwi-run. This option might be useful, if one wants to add some 'fresh' external configurations to a pool, which might

be the results of some other simulation performed earlier during the G42-run, thus replacing e.g. the worst configurations in the pool of 'nwalk' walkers. Of course, then one needs to write a detailed meta-program for running such a complex search (see meta-program section below).

Because of this, one should always explicitly use the 'setwalkcurr' command before calling the mws_module(s), if one wants to use only a subset of walkers instead of all 'nwalk' walkers.
walk1 = 1
walk2 = nwalk

8.5.2 Parameters regarding treatment of the pool of configurations newly generated by multi-walker moves or jumpmoves

After mixing moves or jumpmoves, one usually wants to perform some local minimization before accepting or rejecting the new configuration(s). For this, we need certain parameters controlling these local runs. Note that if we are working inside the sa-loc-run module most such parameters are already selected when setting up this module.

- 'poolminimizationflag' = 1 indicates that after a mixing move resulting in 'nwalkdum' new configurations or after a jumpmove outside of the 'sa-localrun' module a local minimization is supposed to take place.
  poolminimizationflag = 1

- If a jumpmove is called from another module besides local-run-simulated annealing, then one would want to have the option to perform a short minimization right after the move; this is done by setting 'jumpminflag' = 1. Then one needs to set 'jumpminflag' and 'poolminimizationflag' both equal one. Note that such a minimization after jumpmoves always takes place in salocruns; thus we use 'jumpminflag' = 0 for that module.
  jumpminflag = 1

- 'poolgraddescflag' = 1 indicates that after the local minimization a gradient descent shall take place. This option is nowadays part of 'poolmintypeflag0'; thus this variable is inactive.
  poolgraddescflag = 0

- 'poolmintypeflag0' indicates, which kind of local run-algorithm is to be used during the poolminimization as part of the call to the moveclass (-multi and -jump), unless the calling module overrides this choice:

  1. 'poolmintypeflag0' = 0: standard stochastic quench

2. 'poolmintypeflag0' = 1: gradient steepest descent
3. 'poolmintypeflag0' = 2: simplex-method (not yet activated)
4. 'poolmintypeflag0' = 3: line search
5. 'poolmintypeflag0' = 4: powell-method
6. 'poolmintypeflag0' = 5: Monte Carlo run at temperature 'Tlocal'
7. 'poolmintypeflag0' = 6: gradient steepest descent in one step calling an external code (e.g. with GULP, QE)
8. 'poolmintypeflag0' = 7: quench followed by gradient descent in one step
9. 'poolmintypeflag0' = 8: Mone Carlo run at 'Tlocal' followed by gradient descent in one step

poolmintypeflag0 = 7 ADD SECOND MINIMIZATION

- 'poolminsteps0' indicates the length of the quench / Monte Carlo / gradient run, while 'Tlocalpool0' gives the local temperature for the moveclass of the quench (step-size adaptive), and Monte Carlo-run. Note that when calling the poolminimization from within a sa-local-run, then 'salrsteps' replaces 'poolminsteps'. 'poolflrlocal' sets the decrease exponent of the local Monte Carlo run being called in the local run after generating the new configuration. In principle, each walker can have it own temperature for the local run and its own run length, but one usually sets them equal.
poolflrlocal = 1.0d0
do i20 = 0,nwalkmax
poolminsteps0(i20) = 100
Tlocalpool0(i20) = 0.01d0
enddo

- 'poolminsaveflag' = 1 indicates that the result of all of these minimizations are to be saved. Considering how much time one has spent in producing the minima, this often makes sense. But note that after 100000 such minimization procedures (for 'nwalkdum' walkers) we start overwriting the old results. If one wants to enlarge this number, one needs to modify the subroutine that enumerates the minima and provides the appropriate suffixes.
poolminsaveflag = 1

8.5.3 Multi-walker moves

- 'fractmulti(mclflag)' indicates the fraction of mixing moves among the multi-walker moves for interacting walkers. This

value can be different for different moveclasses. Since moveclass
2 is usually used for the local minimizations as default, one
should therefore set 'fractmulti'(2) = 0. Although the moveclass
for multi-walker moves 'moveclassmulti' currently can only
be called from a mwi-type module, it is still better to keep
the setup clean.
fractmulti(1) = 0.0d0
fractmulti(2) = 0.0d0

- 'nmovemult' is the number of possible type of multi-walker
  moves. Only change this entry when a new move has been added
  to 'moveclassmulti'.
  nmovemult = 4

As for the standard moves and jump moves for single walkers, 'initmultmflag(...,...)'
indicates the percentages with which the multi-walker moves are
attempted. Thus, they need to add up to one for each moveclass.
Furthermore, each move generates two new configurations from a starting
pair of configurations.

- 'initmultmflag(...,1)': Merging cell parameters of two configurations
  but keeping atom positions the same
  initmultmflag(1,1) = 0.25d0
  initmultmflag(2,1) = 0.25d0

- 'initmultmflag(...,2)': Merging atom positions for two configurations
  but keeping cell parameters unchanged
  initmultmflag(1,2) = 0.25d0
  initmultmflag(2,2) = 0.25d0

- 'initmultmflag(...,3)': Generating an averaged structure weighted
  by random numbers from two structures
  initmultmflag(1,3) = 0.25d0
  initmultmflag(2,3) = 0.25d0

- 'initmultmflag(...,4)': Joining bottom and top halves of two
  structures
  initmultmflag(1,4) = 0.25d0
  initmultmflag(2,4) = 0.25d0

8.5.4 Limitations on cell parameters and atom-atom distances

The next entries deal with the limitations on the cell volume and
atom distances etc. during the runs.

- 'volmaxabs' gives the maximum volume a cell is allowed to have
  (in Ang**3).
  volmaxabs = 4.0d8

- 'volmaxfact' gives the ratio between the maximum volume and the minimum volume (multiplied by 3) a cell is allowed to have (the standard value for 'volmaxfact' is 10, which gets adjusted in subroutine volumemin as long as 'volrangeflag' or 'volinitflag' are not active). Note that we need a rather large value for 'volmaxfact' (100?), if we prescribe a starting cell of large volume, in order to study porous structures.
  volmaxfact = 10.0d0

- 'volminabs' gives the minimum volume the cell is allowed to have
  volminabs = 1.0d0

- 'lmin' gives the minimum length of a basisvector of the cell
  lmin = 1.0d0

- 'aspratio' gives the maximal ratio in the lengths of two cell vectors. This value might need adjusting when one uses starting cells from experiment.
  aspratio = 3.0d0

- 'angmin' gives the minimum and maximum angle ($180-angmin$) between two cell vectors (in degrees)
  angmin = 45.0d0

- 'angmax' gives the maximum angle of the sum of the three angles in a 3-dim simulation (in radian)
  angmax = 5.58d0

- 'tol2' is the minimal allowed distance between two atoms during the energy calculation
  tol2 = 0.001d0

- 'rectifyflag' = 1 means that after each accepted move that involves the change of the cell, we rectify the cell and make it as compact as possible. (afterwards, the energy is recalculated, in order to deal with boundary effects)
  rectifyflag = 0

8.6 'Nochange': outright rejection of proposed moves

Since the chemical system being studied has to obey certain physical limitations, and since we have some limitations on the computational capabilities available, certain proposed moves during the landscape exploration can be (and usually should be) rejected outright before even an energy computation has been performed (or are rejected during

some early stage of the energy computation). Each such rejection has its own 'nochange' code.

Whenever such a rejection happens, the corresponding entry in the 'nochange'-array is increased, and we can at the end of each phase of the exploration receive a list of such rejected moves, which can be useful in analyzing the performance of the algorithms and in understanding features of the energy landscape.

The following values of 'nochange' can appear:

- 'nochange' = 1: No choice made in moveclass (in moveclss.f)
- 'nochange' = 3: No move found for move 4 (in moveclss.f); No move found for move 11 (in moveclss.f)
- 'nochange' = 4: length of cell vector smaller than 'lmin' (in chckcell.f); cell angle smaller than limit value 'cemax' (in chckcell.f); ratio between two cell length larger than 'aspratio' (in chckcell.f); cell too flat, sum over cosines larger than angmax (in chckcell.f); no move found for move 5 (in moveclss.f); no move found for move 6 (in moveclss.f); no move found for move 14 (in moveclss.f); no move found for move 15 (in moveclss.f); no move found for move 16 (in moveclss.f); no move found for move 17 (in moveclss.f)
- 'nochange' = 6: cell volume larger than 'volmax' (in chckcell.f)
- 'nochange' = 7: cell volume smaller than 'volmin' (in chckcell.f)
- 'nochange' = 8: volume less than volrangebottom (in moveclss.f); volume larger than volrangetop (in moveclss.f); no move found for move 8 (in moveclss.f)
- 'nochange' = 9: No success for move 9 (in moveclss.f)
- 'nochange' = 10: atom distance smaller than 'tol2' (in deltae.f); atom distance smaller than 'tol2' (in deltamadelung.f); atom distance smaller than 'tol2' (in energy.f)
- 'nochange' = 11: cell volume larger than 'volmax' while stoichiometry change active (in chckcell.f); cell volume smaller than 'volmin' while stoichiometry change active (in chckcell.f)
- 'nochange' = 12: negatives freies Volumen (in deltae.f); no success in finding a free space for placing an atom (in holesrch.f); no success for move 10 (in moveclss.f)
- 'nochange' = 13: No move found in move 12 (in moveclss.f)

- 'nochange' = 14: No move found for move 13 (in moveclss.f)
- 'nochange' = 17: distance between atoms too small (in pot.f)
- 'nochange' = 15: no move found for move 1 (in moveclss.f); no move found for move 2 (in moveclss.f); no move found for move 3 (in moveclss.f);
- 'nochange' = 16: free volume negative (in energy.f)
- 'nochange' = 20: volume larger than 'volmax' while stoichiometry change active (in deltae.f); volume smaller than 'volmin' while stoichiometry change active (in deltae.f); volume larger than 'volmax' while stoichiometry change active (in energy.f); volume smaller than 'volmin' while stoichiometry change active (in energy.f)
- 'nochange' = 21: volume smaller than 'volmin' (in deltae.f); volume smaller than 'volmin' (in energy.f)
- 'nochange' = 22: volume larger than 'volmax' (in deltae.f); volume larger than 'volmax' (in energy.f)
- 'nochange' = 30; 'nochange1' = 30: distance in penalty space too small for two configurations for penalty function 3 (in penaltyupdate.f)
- 'nochange' = 31; 'nochange1' = 31: distance in penalty space too small for two configurations for penalty function 4 (in penaltyupdate.f)

9 Flowchart of the program

9.1 General aspects

The main program begins in the file g42prog.f. First the entries in the input.f file are read, and the metascript (commandscript) is generated. This script is written into a dummy-protocol of a run usually called 'reference', together with the entries set in input.f. (In this 'run' only a random starting configuration is generated, the database is loaded in, consistency checks are made in inpclcwr.f, and one energy calculation is performed).

Next, the program looks for entries in the command-script, and then follows this script. Whenever it encounters the command 'gennewconf' or 'loadnewconf', it assumes that a sub-run or a major piece thereof has been finished, and writes an endconfiguration for the current run (three versions: with full coordinates of all building units

and atoms, only atom coordinates, and only coordinates of the centers of mass of the building units), closes the protocol and writes the newly generated configuration into a start configuration file and opens a new protocol. Furthermore, it loads the data from the database and checks whether all parameters are appropriate.

Whenever the command corresponds to calling a module, g42 writes the current configuration into a temp-file and copies this file into an input-file for the module. After the module is finished, an output file for the module is generated (usually in the three versions mentioned above). The next module continues with the last configuration of the current module.

If the command contains parameters, these are read and set at this stage. Of course, this holds in particular for those commands that only set parameters and do not activate a module. Note that some parameters are set directly inside the sub-routine that is called. In this case, the externally assigned parameters are overwritten.

9.2 Metascripts

An important feature of the G42+-program is its metascript capability, the ability to programthe code internally by prescribing or automatically generating a metascript, which gives the order in which various modules and simple commands are called and executed. To control this, we define a flag, 'hvprogramflag'. The simplest way to run G42+, is to write the sequence of commands explicitly. This is okay for e.g. testing purposes or for quick runs combining only a couple of modules, or when you explore a particular starting configuration. However, it becomes quite tiresome, if e.g. the sub-runs are repeated for many different random number sequences. Thus, there exists the possibility to program essentially a 'subroutine' within input.f, which generates the full command script automatically.

To achieve such a programmed sequence of commands, we can also have pre-prepared sets of flexibly constructed commands constituting small internal pieces of code inside input.f, which then automatically generate many commands for complex but nevertheless repetitive pieces of command code. Currently, three such types of command meta-programs are provided, with the choice given by the setting of 'hvprogramflag'.

1. 'hvprogramflag' = 0: sequence of explicit commands listed by hand

2. 'hvprogramflag' = 1: repetition of a sequence of exploration runs for different starting configurations, different pressures, and different seeds. Especially useful for massive landscape exploration and global optimizations

3. 'hvprogramflag' = 2: sequence of setpressure-relaxation-setpressure-... commands, which is used to check whether various starting minima found at one type of pressure remain minima at higher (or lower) pressures.

4. 'hvprogramflag' = 3: sequence of threshold runs for various lids starting from a single starting minimum.

hvprogramflag = 0

Since generating the commands automatically involves the generation of more or less complex character strings, we need to involve many character variables. For simplicity, these are all 80 characters long, and fall into two different groups: hvc80_n (n is an integer 1,2,3,...) for configuration names etc., and hvcsuff_n (n = 1,2,3...) for suffixes (or prefixes), which are added to the root of the configuration names in order to let us know, which part of the exploration is currently active. Finally, there are the 80-character strings for lists of lid-values, pressure values, names of initial configurations, initdata and fixdata. The other character variables with names hvc... represents pieces of stringtext of defined length that have been used in the past to construct the repeated parts of the commands.

Please keep always in mind that several strings will be concatenated with various suffixes and prefixes, both during the generation of commands and during the output phase of various modules. Thus, 1) always use the 'trim' command when calling or concatenating one of the string-variables, and 2) do not choose configuration or initdata names that exceed 30 characters!

Quite generally, the commands used in the metascripts consist basically of some sequence like this: Load new configuration, set new seed for random number generator, set new pressure value, perform simulated annealing sub-run, perform quench sub-run, load new configuration,... An important aspect is the fact that sometimes we need to load in the output configuration of the previous sub-run, in order to use it as the input configuration of the new sub-run. Sometimes, this is already provided for, i.e., the new module automatically uses the last file (! This is sometimes, but not always, identical with the output file of the previous module; thus if one wants to use the output file one has to load it in explicitly !) of the previous module to continue the run.

However, one often wants to have more control, i.e., we want to make sure that we read-in the outputfile of the previous module and not the last one etc.. Thus, we need to think about what the name of the previous outputfile is going to be, and then make sure that we enter this name in the character variable in the command 'loadnewcfg', and similarly, we have to think of giving the new

configuration an appropriate name. In order to achieve this automatically, we use character variables, which are placed after each other using the Fortran '//' syntax. But this requires that we do not have empty spaces in the names, and the names must have exactly the number of characters as the definition demands. In particular, this must also hold true for all the character variables that are constructed from joining several such character-name-pieces! This is one of the two reasons for using the 'trim' command; the other is that we do not want to create character strings that are beyond our capability to handle properly resulting in undesirable cut-offs.

Note that when you create your own character variables, make sure they start with 'hv' (for local 'help variable'), in order to avoid interference with names that have already been used globally in 'varcom.inc'. Furthermore, note that most of the arrays employed have no more than 99 entries. If you want to use arrays with more entries, you will have to change the entries in the 'tform'-function, which converts numbers to character strings, accordingly.

Concerning the actual program set-up, remember that in (primitive) Fortran only 72 columns of a line are taken into account when compiling. The actual executable commands start in line 7, and have to finish by line 72 (unless continued to the next line). The first six columns are employed for specifying line numbers (to serve as targets of 'goto' commands) and numbers of format statements, for indicating that the current line is a comment, or for indicating that the current line is a continuation of the previous line. A 'c' at the beginning of a line (first column) indicates a comment (the line is ignored), and a 'f' in the sixth column of a line indicates that the line is a continuation of the formula from the preceding line (thus allowing to evade the 72 character limitation).

If you do want to add your own complex metascript inside input.f, be careful: Make sure that 'icheck' is correctly calculated. Add the command 'end' at the end. Make sure that the new character variables 'hvc...' are consistent in length and that the 'trim' command is used everywhere. Be careful, not to introduce new global variables: all the variables should be contained inside input.f. Finally, 'tform(in1,length1)' is a function that generates for an integer number 'in1' a character string of length 'length1'. This string that is returned is always padded to be of length 5, but the relevant information is in the first 'length1' characters. When we use a 'smaller' string variable in the calling program to place 'tform' onto, we read off from the beginning. An error occurs, if we send from 'tform' with length 'length1' = 5, but use a string variable with length equal e.g. 4. Always match the 'length1' parameter with the length of the receiving string.

9.2.1 Parameters for meta-program: repeated block of commands

The typical situation consists of small blocks of commands that are repeated for many seeds. This is now usually replaced by running the many seeds via the multi-walker-separate option. But if we need to repeat the same run hundreds or more times, then we cannot do this with a simple multi-walker option anymore since the required memory is too large.

Of course, the kind of repeated set of commands is up to the user, so you will have to insert them on your own inside the script at the appropriate place! Several of the commands that have been used in the past have been left inside the script but commented out ('c' character at the beginning of the line.

The seed we set at the beginning is thus the 'masterseed'. If one wants to repeat multi-walker modules for new 'masterseeds', one can do that, of course. 'commrepflag' gives the number of repeats of a block in the command-array. 'commblk' denotes the length of this block. 'hvstart' is the starting number for the masterseeds and the suffixes of the new name of the generated function. 'hvseeddiff' is the difference between two successive master seeds. The first 'masterseed' is 'hvstart'+1. Make sure that you separate two 'masterseeds' by at least 'nwalk' ('hvseeddiff' $\leq$ 'nwalk'); else you just repeat runs. Note that one might want to expand/modify this initial set-up appropriately for other types of repeated blocks.

```
commrepflag = 0
hvstart = 0000
hvseeddiff = 10
commblk = 5
```

If we want to set 'hvstart' via reading from an external file, then set both 'readinflag1' and readinflag' to one. readinflag1 = 0

Note that if we want to run, e.g., several different 'initdata'-configurations, the loop over these configurations is external to the block of commands described by 'commrepflag' and 'commblk'. 'hvlconf' gives the number of configurations, for which the whole inner block is to be repeated 'commrepflag' times.

You are free to use these array names in your own script program, but in the meta-program(s) provided, they denote the list of root names for the sub-runs ('hv80confname(...)'), the list of files to be loaded in ('hv80initdata(...)'), the list of files with fixed atoms to be loaded in ('hv80fixdata(...)'); furthermore, the list of pressures being investigated ('hv80press(...)') and lids to be studied (hv80thlid(...)). These arrays can contain up to 99 entries each. Note that we need to provide at least the first entry in these arrays, even if we only perform random generation of new files,

```
i.e. we have only one configuration name, for initialization purposes.
Also, even if we only want to use the pressure given in the main
part of input.f, we have to provide it again as the first entry
in 'hv80press' since the meta-script explicitly sets the pressure
according to the array (unless you change the meta-script, of course).
hv80confname(1) = 'cov003-14.1.7.0'
hv80confname(2) = 'chck38.0.8.0.02'
hv80confname(3) = 'chck38.0.8.0.03'

   hv80initdata(1) = 'Test-MgO-Bulk'
hv80initdata(2) = 'db38.0.8.0-02tri8.dat'
hv80initdata(3) = 'db38.0.8.0-03tri8.dat'

   hv80fixdata(1) = 'test'
   'hvlconf' is the number of different configuration names
hvlconf = 1
   Note that if we want to run for several different pressures,
we must enter values into the variable array 'hv80press'. This should
be done in a F-format, so that G42+ can interpret the string. In
order to allow a smooth operation, one should make each entry of
the same length (e.g. 15 characters, including the . and the -sign.).
Remember that pressure is given in units of eV/Ang**3.
hv80press(1) = '0.0000000000000'
hv80press(2) = '10.000000000000'
hv80press(3) = '1.0000000000000'
   hvlpress is the number of different pressures used and 'hvpressoffset'
is the offset in 'hv80press'. This is useful for the purpose of
adding more pressures to the same type of exploration when starting
a new set of runs. This way, you can keep the same naming conventions
where the pressures are included by their position in the 'hv80press'-array
(e.g. pressure = 0 corresponds to pressure-suffix 1, and pressure
= 10 to pressure-suffix 2, respectively).
hvlpress = 1
hvpressoffset = 0
```

9.2.2 Parameters for meta-program: Slowly changing pressures

```
We also can be interested in the following scenario: We have found
a local minimum at some pressure, and now we want to know, whether
it is still a minimum at other pressures. Thus, we construct the
following sequence: Quench-Change Pressure- Load output-Quench-...
We can employ the pressure list above, and the command sequence
given below for 'npress' greater than zero.
   'npress' is the number of pressure steps to be explored, 'hvc80_1'
```

```
is the overall 'confname', 'hvc80_2' is the name of the initial
starting configuration. Note that we use only one seed value, given
by the explicit seed-declaration in input.f. In principle, one could
write a larger metascript with additional loops, that allows for
several quenches with different random seeds at each pressure stage,
of course.
npress = 0
hvc80_1 = 'testconf'
hvc80_2 = 'testinitdata'
```

9.2.3 Parameters for meta-program: Multi-lid threshold runs

```
The set-up of for threshold runs with many stops in-between and
several lids is somewhat more complicated. Of course, we can stop
the threshold runs many times along their trajectories and perform
many quenches, but in this case, we cannot reproduce only some piece
of a run that is of particular interest (e.g. in order to measure
some additional quantity along the way) - We would have to repeat
the whole run if we wanted to do so because of the random number
generator. But if we set it up as a piece-wise procedure with well-defined
random number selections and input-files, then this is possible,
of course. To achieve this, we can use the block of commands below
to generate the sequence of necessary commands automatically.
   The basic scheme is the following: We start from a given initial
minimum, select a lid from the lid-array, run a short threshold-run
below this lid, use the final configuration of this run as starting
point for several quench runs, perform a second threshold run starting
from the final configuration of the first run, etc. For every lid,
we repeat this sequence of run threshold-quench-threshold-... runs
for different random number sequences. Finally, the whole process
is repeated for the next lid in the lid-array.
   'hvc80_1 is the underlying 'confname' during the threshold run,
and 'hvc80_2' is the name of the starting configuration at the beginning
of the first threshold run (usually a local minimum).
hvc80_1 = 'testthreshMgO4'
hvc80_2 = 'no-endcfgtestMgO'
   'nthlids' gives the number of lid values to be explored from
the starting minimum ('hvc80_2'). 'nthlids' should be smaller or
equal to the length of the array 'hv80thlid' where the energies
of the lids are stored.
nthlids = 0
   'hv80thlid' is an array with a sequence of lids to be investigated
by the program. The energies of the lids are given as character
variables in the F-format. They should not be given in exponential
```

```
format.
hv80thlid(1) = '-18.600000'
hv80thlid(2) = '-18.500000'
hv80thlid(3) = '-18.400000'
hv80thlid(4) = '-18.300000'
hv80thlid(5) = '-18.200000'
hv80thlid(6) = '-18.100000'
hv80thlid(7) = '-18.000000'
hv80thlid(8) = '-17.900000'
hv80thlid(9) = '-17.800000'
hv80thlid(10) = '-17.700000'
   'hvlidoffset' gives an offset, i.e., if we want to start the
labeling of lid runs not with the bottom lid, but with a higher
number. This can make sense, if one wants to 'add' higher lids without
having to relabel the output files (e.g. if one has already made
runs with lids one and two, and now starts a new set of runs for
lids three and four). Furthermore, one usually wants to use the
same sequence of lids for many different minima in a system, in
order to match the observed barriers. But since it does not make
sense to perform a lid run at lids (and thus energies) that are
below the starting minimum, one would want to use the same list
of lids but a different offset for each minimum.
hvlidoffset = 0
   'nthseed' is the number of seeds for a given lid value; one wants
to repeat the runs for the same lid and starting minimum, in order
to gain some statistics regarding the landscape. 'hvthseedinit'
is the value of the seed (= masterseed) at the beginning of the
generation loop (analogous to 'hvstart') Note that we always generate
a new seed value at the beginning of another piece of threshold
run. In this way, we can reproduce pieces of the threshold analysis
if desired. In order to take the multi-walker option into account,
we select the difference between subsequent random number seeds
'hvthseeddiff' to be larger or equal to the number of walkers employed.
nthseed = 3
hvthseedinit = 1
hvthseeddiff = 1
   'nthrun' is the number of pieces of a threshold run of individual
length ('looplim' =) 'hvc80_3'. The length 'hvc80_3' is given as
a character string in the I-format.
nthrun = 10
hvc80_3 = '005000'
   'nthquench' is the number of quenches to be started at the end
of each such threshold run piece. The length of the quenches etc.
should be set in the quench-module. Note that the pressure used
```

```
in the run is the one given by 'press = ' in input.f.
nthquench = 5
```

9.3 Metascripts: code of the meta-programs

```
Next, we provide the explicit codes provided in input.f for the
different meta-program options described above. After each command
has been put into the command-array, we also print it to screen
(or nohup.out) in order to gain an overview over what the commands
will be. A complete list of the commands generated is also provided
in the protocol of the test (reference) configuration.
```

9.3.1 Meta-program 0: Explicit list of commands

```
if (hvprogramflag.eq.0) then
command(1) = 'setseed(00002)'
command(2) = 'gennewcfg(test2La2F2Se_QE1,1,1,0,0,1,1)'
command(3) = 'setwalkcurr(1,1)'
command(4) = 'mws_simann'
command(5) = 'mws_graddesc'
command(6) = 'end'
   Set 'icheck' equal to the number of the last command ('end')
icheck = 5
```

9.3.2 Meta-program 1: Script for repeated block of commands

```
The outermost loop is the choice of different root-configuration
names. Usually, different root-configuration names are needed if
we are loading in from a set of dfiferent starting configurations
from the array 'hv80initdata'. Next we have a loop over different
pressures (including the 'setpress' command), and then the loop
over different random numbers. Inside this last loop we have the
actual sequence of commands such as the selection of a new random
seed ('setseed' command), generation or loading of a starting configuration,
setting the number of walkers to be used, and the actual modules
to be run in sequence.
elseif (hvprogramflag.eq.1) then
icheck = (commrepflag*commblk + 1)*hvlconf*hvlpress + 1
if (icheck.gt.commax) then
print *,'too many commands'
stop
endif
commcount = 0
do i7 = 1,hvlconf
do i6 = 1,hvlpress
```

```
hvcsuff_2 = tform(hv3+i6,2)
The suffix 'hvcsuff_2' indicates the pressure being used
commcount = commcount + 1
command(commcount) =
f 'setpress('//trim(hv80press(hvpressoffset+i6))//','
f //trim(hvcsuff_2)//')'
print *,commcount,command(commcount)
hvstart0 = hvstart
do i5 = 1,commrepflag
hv1 = hvstart0+i5*hvseeddiff
hvcsuff_1 = tform(hv1,5) 'hvcsuff_1' is both the 'masterseed' of
the run and one of the suffixes to its name, hv80confname//hvcsuff_1,
giving the full 'confname' (together with the suffix for the pressure
'hvcsuff_2'. Note that the suffix indicating the identity of the
walker is added subsequently during the loading or the new generation
of starting configurations.
   Now comes the start of the repeated block of commands:
commcount = commcount + 1
command(commcount) =
f 'setseed('//trim(hvcsuff_1)//')'
print *,commcount,command(commcount)
hvc80_1 = trim(hv80confname(i7))//'-'//trim(hvcsuff_2)
f //'-'//trim(hvcsuff_1)
The next piece is the (commented out) alternative of loading in
a configuration from the array 'hv80initdata'. Remember that you
need to provide the configurations to be loaded in with already
a suffix '_????', where '????' corresponds to the number of the
walker, e.g. '0001'.
c commcount = commcount + 1
c command(commcount) =
c f 'loadnewcfg('//trim(hvc80_1)//','
c f //trim(hv80initdata(i7))//',1,1,4)'
c print *,commcount,command(commcount)
This piece generates a new random configuration (typical for the
start of a global exploration):
commcount = commcount + 1
command(commcount) =
f 'gennewcfg('//trim(hvc80_1)//',1,1,0,0,1,4)'
print *,commcount,command(commcount)
Next we select the walkers:
commcount = commcount + 1
command(commcount) = setwalkcurr(1,4)
print *,commcount,command(commcount)
Now come the actual module commands; most of them are commented
```

```
out:
c commcount = commcount + 1
c command(commcount) = 'mws_path'
c print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = 'mws_simann'
print *,commcount,command(commcount)
c commcount = commcount + 1
c command(commcount) = 'mws_monte'
c print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = 'mws_quench'
print *,commcount,command(commcount)
   This is the end of the repeated block of commands. Remember:
the numbers of commands in the block that is to be repeated must
be equal to 'commblk' (= 5 in this example: 'setseed', gennewcfg',
'setwalkcurr', mws_simann', mws_quench').
enddo
End of the repeating of the inner command block
enddo
End of the repeating the pressure loop
enddo
End of the run over all the confnames/initdata. Now follows the
final 'end' command:
commcount = commcount + 1
command(commcount) = 'end'
print *,commcount,command(commcount)
Finally, we check that the total number of commands is correct:
if (commcount.ne.icheck) then
print *,commcount,icheck
stop
endif
```

9.3.3 Meta-program: Script for quench-pressure-quench sequence

```
Note that here you have to count yourself the number of commands
that are needed for each pressure (= 4 in this example). Of course,
you can extend this script, e.g. by adding a gradient descent after
the quench module.
elseif (hvprogramflag.eq.2) then
icheck = npress*4 + 1
if (icheck.gt.commax) then
print *,'too many commands'
stop
```

```
endif
commcount = 0
do i7 = 1,npress
hvc80_3 = trim(hv80press(i7+hvpressoffset)
'hvcsuff_1' indicates the number of the pressure in the pressure
array 'hv80press', and is used to specify configuration name.
hvcsuff_1 = tform(i7+hvpressoffset,2)
commcount = commcount+1
Note that we need to make a distinction concerning the file that
gets loaded in: For the first pressure (i7 = 1), we take the initial
data provided from outside G42+, while for the subsequent pressures,
we take the outcome of the local minimization at the previous pressure.
if (i7.eq.1) then
command(commcount) =
f 'loadnewcfg('//trim(hvc80_1)//'-'//trim(hvcsuff_1)//','
f //trim(hvc80_2//',1,1,1)'
else
hvc80_4 = 'no-mwsquoutp'//trim(hv80_5)
command(commcount) =
f 'loadnewcfg('//trim(hvc80_1)//'-'//trim(hvcsuff_1)//','
f //trim(hvc80_4)//',1,1,1)'
endif
print *,commcount,command(commcount)
commcount = commcount + 1
setwalkcurr(1,1)
print *,commcount,command(commcount)
hvc80_5 = trim(hvc80_1)//'-'//trim(hvcsuff_1)
commcount = commcount + 1
command(commcount) = 'setpress('//trim(hvc80_3)//',1)'
print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = 'mws_quench'
print *,commcount,command(commcount)
enddo
End of the loop over all the pressures. Now follows the 'end' command:
commcount = commcount + 1
command(commcount) = 'end'
print ',commcount,command(commcount)
We next check whether the number of command is correct:
if (commcount.ne.icheck) then
print *,commcount,icheck
stop
endif
```

9.3.4 Meta-program: Script for sequence of threshold runs

```
Again keep in mind that you have to compute the number of commands
yourself: Here: for each of the 'nthquench' quenches after a threshold
run piece, we need 5 commands, and in addition each threshold run
piece requires another 4 commands, and there are 'nthrun' threshold
run pieces that are repeated for 'nthseed' random number sequences,
and the whole takes place for 'nthlids' lid values.
elseif (hvprogramflag.eq.3) then
icheck = nthlids*nthseed*nthrun*
f (4 + 5*nthquench) + 1
if (icheck.gt.commax) then
print *,'too many commands'
stop
endif
commcount = 0
seedcur = hvthseedinit - 1
do 1 i6 = 1,nthlids
hvc80_4 = trim(hv80thlid(i6+hvlidoffset))
hvcsuff_1 = tform(i6+hvlidoffset,2)
'hvcsuff_1' indicates which of the lids in the lid-array is being
studied.
do 2 i5 = 1,nthseed
hvcsuff_2 = tform(i5,2)
'hvcsuff_2' indicates which seed is being used as master seed
do 3 i7 = 1,nthrun
hvcsuff_3 = tform(i7,2)
'hvcsuff_3' indicates, which piece of the threshold run is being
performed.
   The next lines corresponds to loading in a new configuration.
This will either be the external configuration (i7 = 1), i.e. the
starting minimum 'hvc80_2', or the output configuration of the previous
threshold run piece, 'hvc80_6'. Remember that the name of the external
starting configuration must be of the form 'hvc80_2'_????, where
???? is the number of the walker that is supposed to use the file
'hvc80_2'_???? as starting minimum. 'hvc80_5' is the 'confname' of
the threshold run piece to be started, whose name indicates that
it is for lid number 'hvcsuff_1', repeat number of threshold 'hvcsuff_2'
and threshold run piece number 'hvcsuff_3'.
hvc80_5 = trim(hvc80_1)//'-'//trim(hvcsuff_1)
f //'-'//trim(hvcsuff_2)//'-'//trim(hvcsuff_3)
commcount = commcount + 1
if (i7.eq.1) then
command(commcount) =
```

```
f 'loadnewcfg('//trim(hvc80_5)//','//trim(hvc80_2)//',1,1,1)'
else
command(commcount) =
f 'loadnewcfg('//trim(hvc80_5)//','//trim(hvc80_6)//',1,1,1)'
endif
print *,commcount,command(commcount)
Now we pick the walkers to be used:
commcount = commcount + 1
command(commcount) = setwalkcurr(1,1)
print *,commcount,command(commcount)
hvc80_6 = 'no-mwsthoutp'//trim(hvc80_5)
Note that the reference to which walker is meant is added automatically
when executing the loadnewcfg command. The previous threshold-section
has just produced 'nwalk' outputs that now serve as inputs for the
next section.
   The next line corresponds to 'setseed(seedcur)'. Remember to
make sure that 'hvseeddiff' is larger or equal to the number of
walkers being used, since 'setseed' changes the master seed.
seedcur = seedcur + hvseeddiff
hvcsuff_4 = tform(seedcur,5)
commcount = commcount + 1
command(commcount) = 'setseed('//trim(hvcsuff_4)//')'
print *,commcount,command(commcount)
The next line corresponds to calling the threshold module, 'thresh(Elimit,looplim)'
commcount = commcount + 1
command(commcount) =
f 'mws_thresh('//trim(hvc80_4)//','//trim(hvc80_3)//')'
print *,commcount,command(commcount)
After the threshold run piece, we perform 'nthquench' local minimizations
(here: long quench followed by gradient descent):
do 4 i8 = 1,nthquench
hvcsuff_5 = tform(i8,2)
'hvcsuff_5' indicates the number of the quench run being performed.
Next we load in the output of the threshold run piece for all walkers,
pick the walkers to be used, set the seed, and perform the local
minimization:
hvc80_7 = trim(hvc80_5)//'-'//trim(hvcsuff_5)
commcount = commcount + 1
command(commcount) =
f 'loadnewcfg('//trim(hvc80_7)//','//trim(hvc80_6)//',1,1,1)'
print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = setwalkcurr(1,1)
print *,commcount,command(commcount)
```

```
seedcur = seedcur + hvseeddiff
hvcsuff_6 = tform(seedcur,5)
commcount = commcount + 1
command(commcount) = 'setseed('//trim(hvcsuff_6)//')'
print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = 'mws_quench'
print *,commcount,command(commcount)
commcount = commcount + 1
command(commcount) = 'mws_graddesc'
print *,commcount,command(commcount)
enddo
End of loop over quenches from end point of threshold piece
enddo
End of loop over pieces of threshold run
enddo
End of loop over repetitions of threshold run enddo End of loop
over lids being explored. Now follows the 'end' command:
commcount = commcount + 1
command(commcount) = 'end'
print *,commcount,command(commcount)
Finally, we check whether the number of commands is correct:
if (commcount.ne.icheck) then
print *,commcount,icheck
stop
endif
```

10 Input by user

There are several ways G42+ receives input from the user:

- Setting of array limits in 'varcom.inc' (c.f. section ...).
- Entering entries in the various databases: defining building groups and atoms (c.f. section ...).
- Providing a starting configuration that is to be loaded in (c.f. section ...).
- Creating the command metascript in the input.f file (c.f. section ...).
- Defining the configurations in the input.f file (c.f. section ...).
- Setting output flags in the input.f file (c.f. section ...).

- Defining the parameters of the various modules in the input.f file, which generate/load configurations and perform explorations (c.f. section ...).
- Selecting the energy functions in the input.f file (c.f. section ...).
- Defining the moveclasses for the random-walk based runs in the input.f file (c.f. section ...).
- Setting parameters for the initialization routines in the input.f file (c.f. section ...).
- Setting additional global parameters for the runs in the input.f file (c.f. section ...).

10.1 Data read-in from external parameter file

The program has the option that a few parameters can be read in from an external datafile, called 'externaldata', in order to avoid having to recompile G42+ again. If the flag 'readinflag' is set to one, 'readinflag' = 1, this file is read in. In input.f, we have at each of those variables which can be be read in, in principle, a second specific flag ('readinflag1',...,'readinflag6') that must be set equal to 1, if one wants to overwrite the value from input.f with the value taken from the external datafile. Note that if the code expects such a value to appear in 'externaldata', then it will crash if not provided with it.

Currently, the following variables can be read in from an external file (these variables will be the same for all walkers!):

1. 'hvdat1' = 'hvstart' (format I6)
2. 'hvdat2' = 'nTmax' (format I9)
3. 'hvdat3' = 'tolhs' (format F10.5)
4. 'hvdat4' = 'volfactor' (format F10.5)
5. 'hvdat5' = 'numsteps' (format I9)
6. 'hvdat6' = eabinitsafe (format F15.5)

Note that once you decide to read in data from an external file, entries for all six variables have to be provided in 'externaldata'. You decide later when filling in input.f, whether you actually want to use them according to the value you give for the second, specific flag.

```
readinflag = 0 if (readinflag.eq.1) then
open (10,file='externaldata')
read (10,1100) hvdat1
1100 format (I6)
read (10,1101) hvdat2
1101 format (I9)
read (10,1102) hvdat3
1102 format (F10.5)
read (10,1102) hvdat4
read (10,1101) hvdat5
read (10,1103) hvdat6
1103 format (F15.5)
close(10)
endif
```

10.2 Data for initial dummy test configuration 'reference'

Originally, only one run (today we would call it a sub-run) was being performed with G42+. Thus, we still can define the name of the initial configuration and the various ways to generate it within input.f without having to use the command 'gennewcfg' or 'loadnewcfg'. Nowadays, we use this initial configuration as a dummy configuration called 'reference', in order to test whether all the parameters, flags, configuration variables, etc. that are entered in input.f are consistent.

Note that most of the variables (e.g. dimensionality of the system: 'dimens'; name of the database with atom-specific information 'atombasename'; maximum number of atoms allowed: 'numatommax'; etc.) set in input.f for the purpose of generating this test configuration are actually global and are not changed again during the many subsequent sub-runs, whenever a new configuration is generated or loaded in from an external file! Only those variables and parameters which are explicitly set as part of a command that calls a run-module or explicitly sets a particular variable can be changed during the run.

We now use the initial set-up of one empty reference run for registering the command-sequence etc. We only look at one walker, setting it equal to 'iwalk' = 0. Also, we usually do not load in a configuration but generate one at random. By setting iwalk = 0 at this stage, we only produce one reference configuration. Note that, we assume that no further modules are applied to this reference configuration: After the test-stage, we proceed always via loadnewcfg or gennewcfg. Because of this, all entries that depend on a walker-index 'iwalk' should also include the value for 'iwalk' = 0, even if later only walkers starting with 'iwalk' $\geq$ 1 are being used in the explorations.

iwalk = 0
   At this stage, we also define the number of active walkers that we intend to employ during out explorations, 'nwalk'. Make sure that ('nwalk' + 'nwalkdum' + 2) $\leq$ 'nwalkmax' (set in 'varcom.inc'). The variable 'nwalkdum', which defines the number of walkers that can be generated during cross-over moves is set together with the multi-walker moves.
nwalk = 1
   NOTE: THE FILES AND ARRAYS OFTEN ARE QUITE BIG SINCE THERE CAN BE MANY WALKERS PRESENT. THUS MAKE SURE TO USE AS SMALL VALUES IN 'VARCOM.INC' FOR THE ARRAY SIZES AS POSSIBLE. THIS TAKES A MINUTE, BUT IT WILL BE WORTHWHILE !

- 'confname' is the variable that serves as the root-designation of a configuration during a sequence of sub-runs. It will be changed whenever you use the 'gennewcfg' or 'loadnewcfg' commands (of course, you can give the same name again, but that will usually only lead to confusion when analyzing the output data). Note that 'confname' is going to be extended by many prefixes and suffixes indicating what kind of configuration file is generated at what stage of the run. Thus it is strongly recommended to use a moderately short name not exceeding thirty characters (the limit is 80, but then there will be unpleasant cut-offs of parts of the name along the way). For consistency purposes, we chose as the name of the test configuration always the name 'reference' as initial 'confname'.
  confname = 'reference'

- 'atombasename' is the file name for the database containing information about atoms. This database must be provided with all necessary entries for the atoms etc. involved. For more information about the set-up of the atomdatabase, see section .... This variable is not changed throughout the sequence of runs set up in the metascript.
  atombasename = 'atomdatabase'

- 'bgbasename' is the file name for the building unit database. This database must be provided with all necessary entries for the building units involved. For more information about the set-up of the bgdatabase, see section .... This variable is not changed later.
  bgbasename = 'bgdatabase'

- guptabasename is the file name for the gupta-interaction database. This variable is not changed later. A file with the specified

name is only needed if both 'guptaflag' and 'readguptaflag' are set equal to one. For more information on the guptadata information, see section ...
guptabasename = 'guptadatabase'

- 'initdata' : File name for a complete initial configuration ready to be loaded in. This variable will be changed when using the 'loadnewcfg' command. Since for the initial test we usually only generate a new configuration, no file with this name needs to be provided.
initdata = 'test.dat'

- 'initdataflag' indicates whether the starting configuration should be generated:

  1. 'initdataflag' = 0 : generate a new configuration from scratch (the usual case for the reference configuration)
  2. 'initdataflag' = 1 : Load in the configuration with the name 'initdata'
  3. 'initdataflag' = 2 : Use configuration that contains fixed atoms called 'fixdata'. Warning: for fixed atoms, the moveclass only allows atom moves; for all other moveclasses 'natomfix' and 'ntypfix' must be equal to zero. If only the cell is to be prescribed, then in 'fixdata' one must set 'natumcur' = 0, but 'numtypes' = number of types. In general: 'natumcur' in the configuration file 'fixdata' must equal the total number of atoms that are to be read in and are fixed. Currently, 'fixdata' is not active, because we now have another possibility to deal with the case of having some atoms reside in fixed positions by assigning values 'fixbg(...)', according to which the atoms of a certain type are not to be changed in their positions. Thus, one should not use 'initdataflag' = 2, and one does not need to provide a file 'fixdata. (As usual, such outdated options are kept for future reference in the code.)

```
fixdata = 'fixed.data'
do i1 = 0,nwalkmax
natomfix(i1) = 0
ntypfix(i1) = 0
do i2 = 1,natommax
fixatom(i1,i2) = 0
enddo
enddo
initdataflag = 0
```

- 'seed' is the initial value for the random number generator
  (must be an integer). This value can (and should) be re-set
  using the command 'setseed'.
  seed = 0001

- 'dimens' is the dimensionality of the system. Currently, only
  three-dimensional simulations are active, although quasi-two-dimensional
  system can be studied (c.f. 'move2dflag').
  dimens = 3

- 'spgroup' sets the space group of the configuration. This information
  is only needed for those sub-runs, where one runs optimizations
  of 'freely variable parameters' (a subset of the actual coordinates
  connected via symmetry operations). The default value is 1,
  which is the value required for the usual calculations without
  explicit symmetry constraints being enforced. In particular,
  the information about the space group is used for restrictions
  on cell moves in order to keep the symmetry. Note that we get
  into trouble if we switch from runs with moveclasses that assume
  no symmetry to those with symmetry and conversely. In principle,
  one can use the combination of moveclasses together with the
  'setwalkcurr' command to write a metascript that allows mixing
  such explorations (one walker with and one without space group),
  but this is no recommended.
  do i1 = 0,nwalkmax
  spgroup(i1) = 1
  enddo

- 'icheck' is the number of commands in the command array. It
  is later to be calculated and defined explicitly when one constructs
  the list of commands in the command array; at this early stage,
  it is only defined as a default value.
  icheck = 0

- Note that every time you call the modules 'gennewcfg' or 'loadnewcfg',
  you define a new 'confname'. Thus, you will be able to distinguish
  the output files among the data you produce. However, if you
  should call the same module several times before generating
  a new 'confname', then outputs might be overwritten (e.g.,
  in a module sequence 'quench - gradient descent - quench -
  gradient descent'). Thus, there exists a module counter 'countmodule',
  which counts how many modules are called since the last creation
  of a new 'confname', and is set back to zero when we call 'gennewcfg'
  or 'loadnewcfg'. Up to 999 modules can be called before a repetition
  occurs. If larger numbers are needed, then one needs to change

the appropriate locations in the code (i.e. wherever 'tform(countmodule,3)' appears). Initially, this module counter is set to zero in input.f.
countmodule = 0

10.3 Output control

The main output of G42+ are: configuration files (of bg-, at-, and bgonly-type), a protocol (possibly in several pieces if one uses a parallelized version of G42+), and sample files of the density of states (DOS) and attempted moves (ATT) generated by the threshold algorithm. Important data put into the protocol are: initial settings of flags and other variables, acceptance ratios of moves and counters of how often moves are rejected outright (and for what reasons), energies at various stages, 'best' configurations' (nowadays mostly superfluous since these configurations are also saved as individual files), error messages and warnings.

Also, keep in mind that one should use the 'nohup'-command to run G42+, such that the file nohup.out gives a list of run-time messages that are not inside the code but provide information where the code is currently, and whether unusual things are happening.

In order to control what is being written by G42+, one can use certain flags for output control.

10.3.1 Output of configuration files

- Under certain circumstances, it might be useful to save all the configuration files as a concatenation of in form of one or a few giant files (at-config, bg-config, no-config), in order to avoid having a thousands of millions of tiny files inside one directory. This can be controlled by the flag 'writeconfflag', where we can distinguish between different kinds of files (and the points at which such files are produced during the runs). For the meaning of the prefixes in the file names, see ... . Note that certain files ('temp', 'im', 'ref', 'sha') are never saved in their standard (no-) format in the big files, since they are always needed for transferring data during a run, or for the comparison between configurations.

  1. 'writeconfflag' = 0: No files saved in big file
  2. 'writeconfflag' = 1: all local minimizations, saved trajectories, etc. for at-, bg- and no-configuration files.
  3. 'writeconfflag' = 2: The same as 1 plus 'start', 'end', 'last', 'inp', 'outp', 'im', etc. for the at- and bg-configuration files.

writeconfflag = 0

- 'writeatflag' indicates where at-type configuration files are to be saved, which only contain the positions of the atoms but no building unit information. In principle, all this information is already contained inside the standard (no-) format, but usually it is helpful to also have the (at-) format files available for easy visualization

  1. 'writeatflag' = 0: No at-configuration files are saved
  2. 'writeatflag' = 1: All at-configuration files are saved

  writeatflag = 1

- 'writebgflag' indicates whether bg-type configurations, where only the position (of the centers) of the building units are given, should be saved. Saving these files is useful when trying to understand the general shape of a structure based on the arrangement of the building units alone.

  1. 'writebgflag' = 0: No bg-config are saved
  2. 'writebgflag' = 1: All bg-config are saved

  writebgflag = 0

- 'saveenergyflag' indicates, under which conditions we recalculate the energy when we save a configuration as a file. Usually, whenever a configuration is written, its energy is again computed, but this can become unnecessarily time-consuming if the energy is computed by an external code on ab initio level. Currently, the setting is:

  1. 'saveenergyflag' = 0: we never recalculate the energy of a no-, at- or bg- file, and instead enter the value of the last energy calculation that has been performed.
  2. 'saveenergyflag' = 1: we recalculate only for no-files, and never for any savecurr or savetraj files
  3. 'saveenergyflag' = 2: we recalculate only for the no-files
  4. 'saveenergyflag' = 3: we recalculate for all files, except for savecurr or savetraj files
  5. 'saveenergyflag' = 4: we recalculate for all files (up to now the default, but a problem for ab initio calculations which take lots of time!)

  saveenergyflag = 1

- Often, one might want to save configurations in alternative file-formats that can be read easily by some external codes. While we can use the convertall2all-program to achieve this in individual cases, it is often more useful to have G42+ do it right away. This is controlled by setting the appropriate flag. If you want additional formats being written, add an appropriately named flag and modify the sub-routine where files in different formats are being written. Furthermore, note that these additional formats are only saved whenever the at-type configuration files are saved by G42+, since the information in the (no-) format is usually only useful within G42+ (note that this might change if we are dealing with file formats that contain molecule-topology information such as prmtop-files in AMBER; however, at least the prmtop-files are too complex to generate within G42). The formats available for GULP calculations are generated directly from GULP when the appropriate 'GULPformatflag' is set (see GULP-section). Note that one can use the convertalltoall-code for certain useful conversions of e.g. an 'at-'file into some other format.

  1. If we want to save files in the XCRYSDEN format, set the flag 'XSFformatflag' = 1.
  2. For files in the XYZ-format, there is the flag 'XYZformatflag' = 1. This is not activated, since the XYZ format is not uniquely defined.

  XSFformatflag = 0
  XYZformatflag = 0

- If the atoms in the backgroundstructure are also to be included in the saved configuration file, then set 'outputbackgroundflag' = 1. Note that currently this option is only active for the files that are saved in the XSFformat. If e.g. the 'slightly relaxed' background structure is of interest as a future background structure for other calculations, then one should extract it directly from the XSF-format file.
  outputbackgroundflag = 1

- 'watchtraj' determines whether every 'savestep' steps the current configuration is registered. Note that for the powell routine, 'savestep' should be rather small. We always start a new trajectory when we enter a new module. Note that the maximum number of files we register for one such trajectory is 99999.
  watchtraj = 0
  savestep = 100

- 'savecurnum' indicates, after how many (simulated annealing) steps the current configuration should be saved (it will be always overwritten when the next configuration is save. 'savecurnum' = 0 means 'No saving'.
  savecurnum = 0

- 'savecurbestnum' indicates, after how many (simulated annealing) steps a local best energy state should be saved. 'savecurbestnum' = 0 means 'No saving'. Currently, savecurbestnum is not active.
  savecurbestnum = 0

10.3.2 Energy output

- 'watch' indicates, whether all Ebest configurations should be written into the protocol ('watch' = 1) or not ('watch' = 0).
  watch = 0

- 'watchenergy' determines whether every 'energystep' steps the current energy is registered (implemented in simann-routine and in montecarlo routine)
  watchenergy = 0
  energystep = 10000000

- 'watchpowell' determines, whether every iteration in the powell-cycle is registered with its energy.
  watchpowell = 0

- 'wrflag1' indicates, after how many steps an update of the run should be written into the protocol.
  wrflag1 = 1000000

- 'plotflag1' indicates, whether xbest should be written in protocol (1 = yes)
  plotflag1 = 0

- 'plotflag2' indicates, whether abest should be written in protocol (1 = yes)
  plotflag2 = 0

- 'plotflag3' indicates, whether the mass of the cell should be written in protocol (1 = yes)
  plotflag3 = 0

- 'plotflag4' indicates, whether Ebest should be written in protocol (1 = yes)
  plotflag4 = 0

- 'plotflag5' indicates, whether 'nochange' (outright rejection of moves) should be printed. This information is often quite useful in analyzing the behavior of the system during the exploration.

  1. 'plotflag5' = 1: yes
  2. 'pltoflag' = 2: Make statistics over rejections

  plotflag5 = 2

- 'plotflag6' indicates, whether 'btabest' and 'bta' should be printed in files ('bta' is the cut-off parameter in the Ewald-type summation of Coulomb energies)
  plotflag6 = 0

- 'plotflag7' indicates whether the acceptance rate is to be printed (after every temperature step in simulated annealing etc.; at the end in quench and threshold)
  plotflag7 = 1

- 'plotflag8' indicates whether the movetype should be printed

  1. 'plotflag8' = 1: in the nohup.out file
  2. 'plotflag8' = 2: in the protocol
  3. 'plotflag8' = 3: both in nohup.out and protocol files

  plotflag8 = 0

- If calls to energies, gradients, 2nd derivatives are to be counted, set 'counterenergiesflag' = 1.
  countenergiesflag = 0

- In order to avoid having gigantic numbers files in some scratch directory of an external machine, we can collect the most important ones (configuration, protocol) into tar-files, by setting 'maketarflag' =1.
  maketarflag = 0

- If we do not use different parameters for different walkers, then we do not need to write the (identical) entries of the various arrays into the protocol for all walkers (since they are identical). This can be controlled by 'writeallinitflag'.

  1. If all entries for all walkers are to be written into the protocol in 'inpcalcwrite.f', then set 'writeallinitflag' = 1.
  2. Else, only the value for 'iwalk' = 0 is written ('writeallinitflag' = 0)

10.4 varcom.inc

In the file 'varcom.inc', we define all the global variables and the sizes of all the global arrays in G42+. This includes maximum values for various system sizes such as number of atoms, building groups etc.. These size parameters need to be adjusted, either reducing them in order to create a smaller executable file, or enlarging them to take larger system sizes or more complicated definitions of the systems into account (e.g. when defining the same atom many times with different properties.)

- 'writeprotmax': Maximal number of write statements stored when running modules (i.e. between calls to subroutine 'writeprot')
  parameter (writeprotmax = 1000)
- 'writeconfmax': Maximal number of stored configurations when running modules (i.e. between calls to subroutine 'writeconf')
  parameter (writeconfmax = 100)
- 'wrfemax': Maximal number of entries needed to convey error information as double precision, integer or character variables (do not change unless you have added long error description statements to the code)
  parameter (wrfemax = 10)
- 'nprocmax': Maximal number of processors for parallel computations (only refering to the G42+ code, not to any external programs)
  parameter (nprocmax = 1)
- 'nwalkmax': Maximal number of walkers
  parameter (nwalkmax = 12)
- 'chargemin' and 'chargemax' give the range of allowed ion charges in arrays.
  parameter (chargemin = -2, chargemax = 4)
- 'natommax' is the maximal number of atoms; 'typemax' is the maximal number of atom types.
  parameter (natommax = 2000, typemax = 3)
- 'mflagmax': Maximum number of types of moves. Currently, the value of 'mflagmax' equals 30+number of jumpmoves (the total number of standard single walker moves is 25).
  'stepsmainmax': Maximal number of steps between temperature changes in simann (needed for arrays of data).
  parameter (mflagmax = 20,stepsmainmax = 10000)

- 'configmax': maximum number of testcells saved in a cell search
  when trying to find periodic cells in a given configuration
  using the subroutine 'findcell';
  'zeilmax' is the maximal number of lines dealt with in the
  search through the databases.
  parameter (configmax = 10,zeilmax = 10000)

- 'groupmax' = 'typemax' * ('chargemax' - 'chargemin' + 1): needed
  to define the maximum number of classes of atoms in the cell
  search algorithm.
  'commax': maximum number of commands that can be executed by
  the metascript in G42+.
  parameter (groupmax = 32,commax = 5000)

- 'EBmax': array size for attempts memorized in an adaptive schedule
  for quench runs, and also the number of temperature changes
  in the stopping criterion no. 3 in simulated annealing.
  'Ebstmax': maximum number of temperature changes involved in
  the stopping criteria nos. 1 and 2 in simulated annealing.
  'Eeqmax': maximum number of 'numEchange' steps in Monte-Carlo
  / simulated annealing with equilibrium detection
  parameter(Ebstmax = 50, EBmax = 500, Eeqmax = 5000)

- 'numkmax': maximum number of vectors in reciprocal space used
  during Ewald summation.
  parameter (numkmax = 100)

- 'mclassnummax': maximum number of different moveclasses available
  to chose from (should be at least 2).
  parameter (mclassnummax = 3)

- 'boxmax': maximum number of bins in sampling the local density
  of states (DOS) in threshold runs.
  parameter (boxmax = 100)

- numchangemax: is the maximum number of 'nochange' indicators
  (should only be changed if new 'nochange' categories are added)
  parameter (numchangemax = 40)

- 'bgtypemax': maximum number of building unit types in the configuration
  'nbgmax': maximum number of building units
  'bgsizemax': maximum size of a building unit
  parameter (bgtypemax = 7,nbgmax = 1000, bgsizemax = 20)

- 'atomintmax': maximum number of interacting atom types (for
  GROMACS, AMBER, etc.)
  parameter (atomintmax = 1000)

- 'bgpairmax': maximum number of neighbor pairs in a building unit
  'bgtripletmax': maximum number of neighbor triplets in a building unit
  'bgquartetmax': maximum number of neighbor quartets in a building unit
  parameter (bgpairmax = 80, bgtripletmax = 1000)
  parameter (bgquartetmax = 500)

- 'bgrotpairmax': maximum number of pairs that can serve as axis-atoms (ca. 'bgsizemax'**2)
  parameter (bgrotpairmax = 1000)

- 'nbgmaxfix': maximum number of building units in a fixed background structure
  'natommaxfix': maximum number of atoms in a fixed background structure
  parameter (nbgmaxfix = 1, natommaxfix = 1)

- 'mqloopmax': maximum number of loops in multi-quench runs
  'mqnummax': maximum number of quenches in a block in multi-quench runs
  parameter (mqloopmax = 1000, mqnummax = 100)

- 'saquenchloopmax': maximum number of times we perform a quench during a simulated annealing run
  parameter (saquenchloopmax = 1000)

- 'ptquenchloopmax': maximum number of times we perform a quench during a parallel tempering run
  parameter (ptquenchloopmax = 1000)

- 'lrloopmax': maximum number of temperature loops in local (shadow) runs in saloc-run module
  'lrnummax': maximum number of e.g. quenches in a block in local (shadow) runs in saloc-run module (essentially the number of reference configurations generated for a fixed temperature).
  parameter (lrloopmax = 100, lrnummax = 1000)

- 'palengthmax': maximum number of points along the path in a prescribed path scenario.
  parameter (palengthmax = 1000)

- 'paloopmax': maximum number of loops in a prescribed path scenario.
  parameter (paloopmax = 1)

- 'tempmax' is the maximal size of the temperature array in Monte Carlo or parallel tempering runs; usually this will be a small number - else one would use a rule to generate temperatures automatically.
  parameter (tempmax = 10)

- 'pressmax': size of the pressure array in parallel tempering runs; usually a small number - else one would use a rule to generate pressures automatically. Note that for parallel tempering at the moment one might use a big value, where all columns are the same, if one does not read the temperatures from an array. If both temperature and pressure are read from an array, then 'tempmax' should be equal to 'pressmax'.
  parameter (pressmax = 10)

- 'detsea_nmax': maximum number of variables in the (meta)powell-routine. This value should be larger or equal to (6 + 3 * (number of atoms - 1)).
  parameter (detsea_nmax = 200)

- 'Npairtypemax': maximum number of pairs of atomtypes in a distance-bin-routine. It should be equal or larger than 'typemax' * ('typemax' + 1) / 2
  parameter (Npairtypemax = 6)

- 'Ndistbinmax': maximum number of bins in a distance-bin routine. Note that the first and last bins are overflow due to non-precise atom-atom distances. Thus one should add always 3 more bins to the value one gets from the formula 'Distbinmax' / 'distbinsize' (the third bin is for overflow-safety reasons).
  parameter (Ndistbinmax = 103)

- 'Npenstructmax': maximum number of structures that can be forbidden via a penalty term.
  'Npenpairmax': maximum number of pairs of such structures excluding pairs with itself, thus 'Npenpairmax' = 'Npenstructmax' * ('Npenstructmax' - 1) / 2.
  parameter (Npenstructmax = 501)
  parameter (Npenpairmax = 260000)

- 'Nlaststructmax': maximum number of structures that can be included in the 'distancelast' evaluation. It should be at least as large as 'icheck' in input.f.
  parameter (Nlaststructmax = 600)

- 'SIMPLMPmax' and 'SIMPLNPmax' are parameters in the SIMPLEX routine (currently not active).
  parameter (SIMPLMPmax = 201,SIMPLNPmax = 200)
- 'linescrystmax': maximum number of invariable lines in a CRYSTAL-input.
  parameter (linescrystmax = 200)
- 'N_zlayermax': Maximum number of fixed z-layers in structure
  parameter (n_zlayermax = 1)

CHECK FOR NEW MAX-PARAMETERS
CHECK THE DATABASE ENTRIES

10.5 databases

10.5.1 atomdatabase

In the file 'atomdatabasemaster', all the atoms are listed with their atomic properties relevant for the calculations. Usually, we copy the data for the chemical system of interest into a smaller file named 'atomdatabase' to speed up the runs (the reloading of the data for each new sub-run is much faster than having to scan the full database). Note that all the atoms included are repeated with a 100, 200 etc. added. This is for certain ab initio calculations, where the atom numbers are needed again when using certain pseudo-potentials.

Also, we include the following pseudo-atoms: Voids of different sizes (Hohlraumgroesse) that mimic e.g. templates. Vacancies (placeholders in structures for disordered or ordered atom arrangements). Electron bond. Electron atom consisting of one or more electrons inside a spherical 'bag'. These pseudo-atoms are listed with numbers 1000 and beyond. Note that these number ranges are needed for G42+ to recognize that certain atoms are in reality pseudo-atoms and should be treated differently from standard atoms.

Concerning the entries, the atom mass is given in atomic units ($m_C/12$), the chemical potential is the enthalpy of formation at standard conditions when available, melting and boiling points are all in eV/atom, the ionization energies are in eV/atom, the electron affinities (in eV/atom) have been estimated if no data were available, and the radii in Angstrom are taken from Emsley (The Elements) and are interpolated if some valences were missing.

The names of the entries vary between German and English. In the following, we show a number of example entries. If you want to add your own entriesto the file 'atomdatabasemaster', follow these examples, and, if necessary, check with the readinfo.f and getinfo.f Fortran files. Quite generally, make sure that you not only use the correct sequence of entries, but also that the same

rightward shifts (in units of two empty characters) are included (these are indicated by dashes '-' in this manual). These are crucial for the read-sub-routines to recognize which level of the search hierarchy the read-routine has reached.

- 1.0
  --Wasserstoff
  --Chemisches potential <chempot> = 2.257
  --Atommasse <mass> = 1.008
  --Schmelzpunkt <tmelt> = 0.0012
  --Siedepunkt <tboil> = 0.00175
  --Minimum Ladung <minch> = -1.0
  --Maximum Ladung <maxch> = 1.0
  --Ladung <chrg> = -1.0
  ----Radius <rad> = 1.54
  ----Ionisierungsenergie <ion> = -0.754
  --Ladung <chrg> = 0.0
  ----Radius <rad> = 0.78
  ----Ionisierungsenergie <ion> = 0.0
  --Ladung <chrg> = 1.0
  ----Radius <rad> = 0.01
  ----Ionisierungsenergie <ion> = 13.595

- 1101.0
  --HohlraumGroesse1(HG1)
  --Chemisches potential <chempot> = 0.0
  --Atommasse <mass> = 0.0
  --Schmelzpunkt <tmelt> = 10.0
  --Siedepunkt <tboil> = 10.0
  --Minimum Ladung <minch> = 0.0
  --Maximum Ladung <maxch> = 0.0
  --Ladung <chrg> = 0.0
  ----Radius <rad> = 1.0
  ----Ionisierungsenergie <ion> = 0.0

- 1150.0
  --Vacancy (size 0)
  --Chemisches potential <chempot> = 0.0
  --Atommasse <mass> = 0.0
  --Schmelzpunkt <tmelt> = 10.0
  --Siedepunkt <tboil> = 10.0
  --Minimum Ladung <minch> = 0.0
  --Maximum Ladung <maxch> = 0.0
  --Ladung <chrg> = 0.0
  ----Radius <rad> = 0.01

----Ionisierungsenergie <ion> = 0.0

- This electron bond corresponds to charges in certain regions of a molecule. The actual charge is associated with this 'potential electron' in the building unit.
  1900.0
  --Elektronen-bond (used to associate charge with regions in a molecule)
  --Chemisches potential (standard bildungsenthalpie) <chempot> = 0.0
  --Atommasse <mass> = 0.0005
  --Schmelzpunkt <tmelt> = 0.0
  --Siedepunkt <tboil> = 0.0
  --Minimum Ladung <minch> = 0.0
  --Maximum Ladung <maxch> = 0.0
  --Ladung <chrg> = 0.0
  ----Radius (geschaetzt) <rad> = 0.1
  ----Ionisierungsenergie (geschaetzt) <ion> = 0.0

- This electron bond is a charge in potentia used to model covalent bonds. The charge is associated with this 'potential electron' in the building unit.
  1950.0
  --Electron available for bond (used to model covalent bonds)
  --Chemisches potential (standard bildungsenthalpie) <chempot> = 0.0
  --Atommasse <mass> = 0.0005
  --Schmelzpunkt <tmelt> = 0.0
  --Siedepunkt <tboil> = 0.0
  --Minimum Ladung <minch> = 0.0
  --Maximum Ladung <maxch> = 0.0
  --Ladung <chrg> = 0.0
  ----Radius (geschaetzt) <rad> = 0.001
  ----Ionisierungsenergie (geschaetzt) <ion> = 0.0

- This 'electron atom' corresponds to a 'movable bag' containing between zero and ten electrons. It is used for descriptions of systems with 'free' electrons. The 'ionization energy' is zero, since this 'electron atom' has its own energy terms with an internal Coulomb self energy and a Pauli-energy. Typically, one would allow the radius of the 'electron bag' to vary during the exploration, in addition to the change of its charge.
  2000.0
  --Elektronenatom, i.e. region in space occupied by electrons
  --Chemisches potential (standard bildungsenthalpie) <chempot>

```
= 0.0
--Atommasse <mass> = 0.0005
--Schmelzpunkt <tmelt> = 0.0
--Siedepunkt <tboil> = 0.0
--Minimum Ladung <minch> = -10.0
--Maximum Ladung <maxch> = 0.0
--Ladung <chrg> = -10.0
----Radius (geschaetzt) <rad> = 2.2
----Ionisierungsenergie (geschaetzt) <ion> = 0.0
--Ladung <chrg> = -9.0
----Radius (geschaetzt) <rad> = 2.1
----Ionisierungsenergie <ion> = 0.0
--Ladung <chrg> = -8.0
----Radius <rad> = 2.0
----Ionisierungsenergie <ion> = 0.0
--Ladung <chrg> = -7.0
----Radius (geschaetzt) <rad> = 1.9
----Ionisierungsenergie (geschaetzt) <ion> = 0.0
--Ladung <chrg> = -6.0
----Radius (geschaetzt) <rad> = 1.8
----Ionisierungsenergie (geschaetzt) <ion> = 0.0
--Ladung <chrg> = -5.0
----Radius (geschaetzt) <rad> = 1.7
----Ionisierungsenergie <ion> = 0.0
--Ladung <chrg> = -4.0
----Radius <rad> = 1.5
----Ionisierungsenergie <ion> = 0.0
--Ladung <chrg> = -3.0
----Radius (geschaetzt) <rad> = 1.3
----Ionisierungsenergie (geschaetzt) <ion> = 0.0
--Ladung <chrg> = -2.0
----Radius (geschaetzt) <rad> = 1.0
----Ionisierungsenergie (geschaetzt) <ion> = 0.0
--Ladung <chrg> = -1.0
----Radius (geschaetzt) <rad> = 0.7
----Ionisierungsenergie <ion> = 0.0
--Ladung <chrg> = 0.0
----Radius <rad> = 0.01
----Ionisierungsenergie <ion> = 0.0
```

10.5.2 bgdatabase

In the file 'bgdatabasemaster', all the building groups (building units) are listed with their constituents (atoms, electron bonds,

etc.) and their spatial and charge arrangement relevant for the calculations. Usually, we copy the data for the chemical system of interest into a smaller file named 'bgdatabase' to speed up the runs (the reloading of the data for each new sub-run is much faster than having to scan the full database). Also, we include a number of special pseudo-atom types.

Note that compared to the earlier version of 'bgdatabasemaster', which only allowed for rigid molecules, we have now flexible molecules, where we can enter in addition topological information about established atom pairs, triplets and quartets (and whether the bond distances and angles can be modified during the exploration), information about the 'atom interaction type' needed for external molecule codes like GROMACS or AMBER, and the magnetization of the atoms.

The names of the entries vary between German and English. In the following, we show a number of example entries. If you want to add your own entries to the 'bgdatabasemaster', follow these examples, and, if necessary, check with the readinfo.f and getinfobg.f Fortran files. Like in 'atomdatabasemaster', do not forget the hierarchy-determining pairs of 'empty spaces' (here shown as pairs of dashes).

Note that all atoms have relative coordinates via Cartesian position vectors with respect to a reference point. Try to choose for reference either the 'center of mass' of the building unit, or at least an atom close to the center. Also, place this atom, or (in the case of the reference point being the center of mass but with no atom at this place) the atom closest to the center of mass, at the first position of the atom list of the building unit. The chemical potential for a building unit is set equal to zero throughout, since it is not quite clear, what number one would have to enter here.

The charges and radii entered here override other declarations in the 'atomdatabase' except for building groups that represent individual atoms (Size of building unit <unitsize> = 1.0). Unless we are dealing with individual atoms, we cannot change the charges of the building unit during the simulation run.

Note that if you only want to use the building unit as a rigid one, or if you want G42+ try to figure out the topology of the building unit by itself (this option is not yet active - use the program 'convertall2all' instead to generate the toplogy and add this information to the entry in 'bgdatabasemaster'!), then you set '<pairsize>', '<tripletsize>', and '<quartetsize>'. Thus, e.g. '<pairsize>' = 0 does not mean that one is not able to define 'bonds' - one just does not need them for the calculations intended.

We now present a couple of typical examples of building units:

- 1.0

```
--Hydrogen-atom
--Creation energy <chembgpot> = 0.0
--Size of building unit <unitsize> = 1.0
--Pairsize of building unit <pairsize> = 0.0
--Tripletsize of building unit <tripletsize> = 0.0
--Quartetsize of building unit <quartetsize> = 0.0
--Description level <level> = 1.0
----Atom <number> = 1.0
------Element number <bgatom> = 1.0
------Radius <bgrad> = 0.78
------Charge <chrg> = 0.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
```

- This vacancy is a void, i.e. an empty forbidden zone of radius 1 Angstrom

```
1101.0
--Vacancy size 1 (HG1)
--Creation energy <chembgpot> = 0.0
--Size of building unit <unitsize> = 1.0
--Pairsize of building unit <pairsize> = 0.0
--Tripletsize of building unit <tripletsize> = 0.0
--Quartetsize of building unit <quartetsize> = 0.0
--Description level <level> = 1.0
----Atom <number> = 1.0
------Element number <bgatom> = 1101.0
------Radius <bgrad> = 1.00
------Charge <chrg> = 0.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
```

- This is a real (infinitesimally small) vacancy, not a forbidden zone like the vacancies of size 1 - 7 Angstrom

```
1150.0
--Vacancy size 0 (HG0)
--Creation energy <chembgpot> = 0.0
--Size of building unit <unitsize> = 1.0
--Pairsize of building unit <pairsize> = 0.0
```

--Tripletsize of building unit <tripletsize> = 0.0
--Quartetsize of building unit <quartetsize> = 0.0
--Description level <level> = 1.0
----Atom <number> = 1.0
------Element number <bgatom> = 1150.0
------Radius <bgrad> = 0.01
------Charge <chrg> = 0.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0

- Electron Pseudo-atom 1160.0
  --Electron-pseudo-atom
  --Creation energy <chembgpot> = 0.0
  --Size of building unit <unitsize> = 1.0
  --Pairsize of building unit <pairsize> = 0.0
  --Tripletsize of building unit <tripletsize> = 0.0
  --Quartetsize of building unit <quartetsize> = 0.0
  --Description level <level> = 1.0
  ----Atom <number> = 1.0
  ------Element number <bgatom> = 2000.0
  ------Radius <bgrad> = 0.01
  ------Charge <chrg> = 0.0
  ------Magnetization <magnetization> = 0.0
  ------Atominteractionpotentialtype <atominttype> = 0.0
  ------X-Coordinate <ux> = 0.0
  ------Y-Coordinate <uy> = 0.0
  ------Z-Coordinate <uz> = 0.0

- This is a rigid complex NO3(1-) ion, where we have associated the nitrogen atom with a charge of +5, and all the oxygen atoms with a charge of -2.
  1201.0
  --NO3-unit (5,-2,-2,-2)
  --Creation energy <chembgpot> = 0.0
  --Size of building unit <unitsize> = 4.0
  --Pairsize of building unit <pairsize> = 0.0
  --Tripletsize of building unit <tripletsize> = 0.0
  --Quartetsize of building unit <quartetsize> = 0.0
  --Description level <level> = 1.0
  ----Atom <number> = 1.0
  ------Element number <bgatom> = 7.0

```
------Radius <bgrad> = 0.05
------Charge <chrg> = 5.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
----Atom <number> = 2.0
------Element number <bgatom> = 8.0
------Radius <bgrad> = 1.4
------Charge <chrg> = -2.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 1.2
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
----Atom <number> = 3.0
------Element number <bgatom> = 8.0
------Radius <bgrad> = 1.4
------Charge <chrg> = -2.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = -0.6
------Y-Coordinate <uy> = 1.039
------Z-Coordinate <uz> = 0.0
----Atom <number> = 4.0
------Element number <bgatom> = 8.0
------Radius <bgrad> = 1.4
------Charge <chrg> = -2.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = -0.6
------Y-Coordinate <uy> = -1.039
------Z-Coordinate <uz> = 0.0
```

- Rigid oxygen-molecule with quadrupole-like charge distribution 1204.7

```
--O2-molecule (distance as in O2; +0.5,-1,+0.5; vdW-radius)
--Creation energy <chembgpot> = 0.0
--Size of building unit <unitsize> = 3.0
--Pairsize of building unit <pairsize> = 0.0
--Tripletsize of building unit <tripletsize> = 0.0
--Quartetsize of building unit <quartetsize> = 0.0
--Description level <level> = 1.0
```

```
----Atom <number> = 1.0
------Element number <bgatom> = 8.0
------Radius <bgrad> = 1.4
------Charge <chrg> = 0.5
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.6
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
----Atom <number> = 2.0
------Element number <bgatom> = 8.0
------Radius <bgrad> = 1.4
------Charge <chrg> = 0.5
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = -0.6
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
----Atom <number> = 3.0
------Element number <bgatom> = 1900.0
------Radius <bgrad> = 0.1
------Charge <chrg> = -1.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
```

- This is a rigid tin(II) ion with a steric electron pair 1217.2

```
--Sn+electronpair (charge: +2,0)
--Creation energy <chembgpot> = 0.0
--Size of building unit <unitsize> = 2.0
--Pairsize of building unit <pairsize> = 0.0
--Tripletsize of building unit <tripletsize> = 0.0
--Quartetsize of building unit <quartetsize> = 0.0
--Description level <level> = 1.0
----Atom <number> = 1.0
------Element number <bgatom> = 50.0
------Radius <bgrad> = 0.93
------Charge <chrg> = 2.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
```

```
------Z-Coordinate <uz> = 0.0
----Atom <number> = 2.0
------Element number <bgatom> = 1900.0
------Radius <bgrad> = 1.40
------Charge <chrg> = 0.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 0.0
------X-Coordinate <ux> = 0.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 2.14
```

- This is a flexible test-water molecule (atom distances and angles are far from optimal). Both atom pairs (bond distance) and the triplet (angle) are free to change ('<freepair>' = 1.0, and '<freetriplet>' = 1.0, respectively.) Furthermore, we list for each atom pair (numbered consecutively via '<pairnumber>') both participating atoms given according to their numbers in the list of atoms, and also the 'ideal' bond distance. In this test molecule, the bond distance is only a rough number. The analogous holds true for the description of each triplet: They are numbered by '<tripletnumber>', and we list the three atoms ABC (identified by their numbers in the atom list) in such an order that the angle is about the atom B in the middle. For this to make sense, one should always have bonds AB and BC. Finally, we proceed analogously with the quartets of atoms ABCD, where the order in which they are listed should be such that a sensible dihedral angle is defined by these four atoms, and there exist reasonable bonds AB, BC and CD between them.
  2002.0
  --H2O-test-molecule
  --Creation energy <chembgpot> = 0.0
  --Size of building unit <unitsize> = 3.0
  --Pairsize of building unit <pairsize> = 2.0
  --Tripletsize of building unit <tripletsize> = 1.0
  --Quartetsize of building unit <quartetsize> = 0.0
  --Description level <level> = 1.0
  ----Atom <number> = 1.0
  ------Element number <bgatom> = 8.0
  ------Radius <bgrad> = 1.40
  ------Charge <chrg> = -2.0
  ------Magnetization <magnetization> = 0.0
  ------Atominteractionpotentialtype <atominttype> = 81.0
  ------X-Coordinate <ux> = 0.0
  ------Y-Coordinate <uy> = 0.0

```
------Z-Coordinate <uz> = 0.0
----Atom <number> = 2.0
------Element number <bgatom> = 1.0
------Radius <bgrad> = 0.1
------Charge <chrg> = 1.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 10.0
------X-Coordinate <ux> = 1.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
----Atom <number> = 3.0
------Element number <bgatom> = 1.0
------Radius <bgrad> = 0.1
------Charge <chrg> = 1.0
------Magnetization <magnetization> = 0.0
------Atominteractionpotentialtype <atominttype> = 10.0
------X-Coordinate <ux> = -1.0
------Y-Coordinate <uy> = 0.0
------Z-Coordinate <uz> = 0.0
--Description level <level> = 2.0
----Pair number <pairnumber> = 1.0
------First pair atom <pairatom1> = 1.0
------Second pair atom <pairatom2> = 2.0
------Ideal distance <distance> = 1.0
------Freepairflag <freepair> = 1.0
----Pair number <pairnumber> = 2.0
------First pair atom <pairatom1> = 1.0
------Second pair atom <pairatom2> = 3.0
------Ideal distance <distance> = 1.0
------Freepairflag <freepair> = 1.0
--Description level <level> = 3.0
----Triplet number <tripletnumber> = 1.0
------First triplet atom <tripletatom1> = 2.0
------Second triplet atom <tripletatom2> = 1.0
------Third triplet atom <tripletatom3> = 3.0
------Ideal angle <angle> = 180.0
------Freetripletflag <freetriplet> = 1.0
```

- The description of flexible molecules with enough atoms to allow for dihedral angles, or quite generally one or more quartets of atoms, is analogous to the case of the molecules with triplets of atoms. The information regarding the dihedral angles appears in the data hierarchy under the heading Description level <level> = 4.0 as

```
--Description level <level> = 3.0
----Quartet number <quartetnumber> = 1.0
------First quartet atom <quartetatom1> = 2.0
------Second quartet atom <quartetatom2> = 1.0
------Third quartet atom <quartetatom3> = 3.0
------Fourth quartet atom <quartetatom4> = 4.0
------Ideal angle <angle> = 180.0
------Freequartetflag <freequartet> = 1.0
```

10.5.3 guptadatabase

In the file 'gupatdatabasemaster', all the atoms are listed with their atomic properties relevant for a Gupta-potential calculation. Usually, we copy the data for the chemical system of interest into a smaller file named 'guptadatabase' to speed up the runs (the reloading of the data for each new sub-run is much faster than having to scan the full database). Note that here we need the names of partners, i.e., for each atom, we have a series of partner-atoms, for which the coefficients in the Gupta-potential are given. Also, the data for the Gupta-potential are taken from fits to experiment, which does pose some problem regarding their validity on the whole energy landscape.

The names of the entries vary between German and English. In the following, we show a number of example entries. If you want to add your own entries to the file 'guptadatabasemaster', follow these examples, and, if necessary, check with the readinfo.f and getinfo.f Fortran files. The entries are quite analogous to the ones in 'atomdatabase'. The charges are set to zero since they are meaningless in the Gupta-potential.

- ```
  13.0
  Aluminium
  Chemisches Potential <chempot> = 3.333
  Atommasse <mass> = 26.98
  Schmelzpunkt <tmelt> = 0.08
  Siedepunkt <tboil> = 0.236
  Minimum Ladung <minch> = 0.0
  Maximum Ladung <maxch> = 0.0
  Ladung <chrg> = 0.0
  Radius <rad> = 1.43
  Ionisierungsenergie <ion> = 0.0
  Potential <pot> (8.0 entspr. Gupta) = 8.0
  Partner <part> (Al) = 13.0
  Ladung des Partners <ldpart> = 0.0
  ```
  ```

```
Agupta <agup> (aus PRB 48:22) = 0.1221
Qgupta <qgup> (aus PRB 48:22) = 2.516
Pgupta <pgup> (aus PRB 48:22) = 8.612
Zgupta <zgup> (aus PRB 48:22) = 1.316
Rgupta <rgup> (aus PRB 48:22) = 2.864
```

- ```
  28.0
  Nickel
  Chemisches Potential <chempot> = 3.851
  Atommasse <mass> = 58.69
  Schmelzpunkt <tmelt> = 0.149
  Siedepunkt <tboil> = 0.259
  Minimum Ladung <minch> = 0.0
  Maximum Ladung <maxch> = 0.0
  Ladung <chrg> = 0.0
  Radius <rad> = 1.24
  Ionisierungsenergie <ion> = 0.0
  Potential <pot> (8.0 entspr. Gupta) = 8.0
  Partner <part> (Ni) = 28.0
  Ladung des Partners <ldpart> = 0.0
  Agupta <agup> (aus PRB 48:22) = 0.0376
  Qgupta <qgup> (aus PRB 48:22) = 1.189
  Pgupta <pgup> (aus PRB 48:22) = 16.999
  Zgupta <zgup> (aus PRB 48:22) = 1.070
  Rgupta <rgup> (aus PRB 48:22) = 2.491
  Partner <part> (Al) = 13.0
  Ladung des Partners <ldpart> = 0.0
  Agupta <agup> (aus PRB 48:22) = 0.0563
  Qgupta <qgup> (aus PRB 48:22) = 1.2823
  Pgupta <pgup> (aus PRB 48:22) = 14.997
  Zgupta <zgup> (aus PRB 48:22) = 1.2349
  Rgupta <rgup> (aus PRB 48:22) = 0.0
  ```

Note that due to the non-transferability of the Gupta-parameters between chemical systems, and the need to refit them for every pair of atom types, it is debatable whether a 'guptadatabase' is as useful as the 'atomdatabase' it is modeled on. Furthermore, we might want to employ smooth cut-offs of the Gupta-potential, which need to be defined explicitly. Finally, in order to be able to combine the Gupta-calculations with other potentials, one should include the gupta interactions as part of this input.Thus, we have the option to enter all the parameters relevant for the Gupta-potential calculations directly inside input.f.

We now present the Gupta-parameter input inside input.f, plus the spline information.
  ```

```
Agup(1,1) = 0.0d0
Agup(1,2) = 0.0d0
Agup(2,1) = 0.0d0
Agup(2,2) = 0.0d0
Pgup(1,1) = 0.0d0
Pgup(1,2) = 0.0d0
Pgup(2,1) = 0.0d0
Pgup(2,2) = 0.0d0
Rgup(1,1) = 0.0d0
Rgup(1,2) = 0.0d0
Rgup(2,1) = 0.0d0
Rgup(2,2) = 0.0d0
Zgup(1,1) = 0.0d0
Zgup(1,2) = 0.0d0
Zgup(2,1) = 0.0d0
Zgup(2,2) = 0.0d0
Qgup(1,1) = 0.0d0
Qgup(1,2) = 0.0d0
Qgup(2,1) = 0.0d0
Qgup(2,2) = 0.0d0
If we use the spline-cutoff version of the Gupta-potential, we need
to add values for the two cutoff-points (i.e. end of standard Gupta-potential,
and the end of spline-part at VGupta = 0) for each pair of atomtypes
(make sure that rc(i,j) = rc(j,i))
rc1(1,1) = 4.0d0
rc1(1,2) = 4.0d0
rc1(2,1) = 4.0d0
rc1(2,2) = 4.0d0
rc2(1,1) = 5.0d0
rc2(1,2) = 5.0d0
rc2(2,1) = 5.0d0
rc2(2,2) = 5.0d0
```