USER'S GUIDE

SIESTA 1.2.15

July 14, 2002

Emilio Artacho	*University of Cambridge*
Julian D. Gale	*Imperial College, London*
Alberto García	*Universidad del País Vasco, Bilbao*
Javier Junquera	*Université de Liège, Belgium*
Richard M. Martin	*University of Illinois at Urbana-Champaign*
Pablo Ordejón	*Institut de Ciència de Materiales, CSIC, Barcelona*
Daniel Sánchez-Portal	*Universidad del País Vasco, San Sebastián*
José M. Soler	*Universidad Autónoma de Madrid*

`http://www.uam.es/siesta`

`siesta@wm.lc.ehu.es`

Contents

1 INTRODUCTION

Siesta (Spanish Initiative for Electronic Simulations with Thousands of Atoms) is both a method and its computer program implementation, to perform electronic structure calculations and *ab initio* molecular dynamics simulations of molecules and solids. Its main characteristics are:

- It uses the standard Kohn-Sham selfconsistent density functional method in the local density (LDA-LSD) or generalized gradient (GGA) approximations.
- Uses norm-conserving pseudopotentials in its fully nonlocal (Kleinman-Bylander) form.
- The basis set is a very general and flexible linear combination of numerical atomic orbitals (LCAO). It allows arbitrary angular momenta, multiple-zeta, polarized and off-site orbitals.
- Projects the electron wavefunctions and density onto a real-space grid in order to calculate the Hartree and exchange-correlation potentials and their matrix elements.
- Besides the standard Rayleigh-Ritz eigenstate method, it allows the use of localized linear combinations of the occupied orbitals (valence-bond or Wannier-like functions), making the computer time and memory scale linearly with the number of atoms. Simulations with several hundred atoms are feasible with modest workstations.
- It is written in Fortran 90. In particular, memory is allocated dynamically, so there is no need to recompile the program when the problem size changes.
- It may be compiled for serial or parallel execution (under MPI). (Note: This feature might not be available in all distributions.)

It routinely provides:

- Total and partial energies.
- Atomic forces.
- Stress tensor.
- Electric dipole moment.
- Atomic, orbital and bond populations (Mulliken).
- Electron density.

And also (though not all options are compatible):

- Geometry relaxation, fixed or variable cell.
- Constant-temperature molecular dynamics (Nose thermostat).
- Variable cell dynamics (Parrinello-Rahman).

- Spin polarized calculations (collinear or not).
- k-sampling of the Brillouin zone.
- Local and orbital-projected density of states.
- Band structure.

References:

- "Unconstrained minimization approach for electronic computations that scales linearly with system size" P. Ordejón, D. A. Drabold, M. P. Grumbach and R. M. Martin, Phys. Rev. B **48**, 14646 (1993); "Linear system-size methods for electronic-structure calculations" Phys. Rev. B **51** 1456 (1995), and references therein.

  Description of the order-N eigensolvers implemented in this code.

- "Self-consistent order-N density-functional calculations for very large systems" P. Ordejón, E. Artacho and J. M. Soler, Phys. Rev. B **53**, 10441, (1996).

  Description of a previous version of this methodology.

- "Density functional method for very large systems with LCAO basis sets" D. Sánchez-Portal, P. Ordejón, E. Artacho and J. M. Soler, Int. J. Quantum Chem., **65**, 453 (1997).

  Description of the present method and code.

- "Linear-scaling ab-initio calulations for large and complex systems" E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

  Description of the numerical atomic orbitals (NAOs) most commonly used in the code, and brief review of applications as of March 1999.

- "Numerical atomic orbitals for linear-scaling calculations" J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B **64**, 235111, (2001).

  Improved, soft-confined NAOs.

- "The Siesta method for ab initio order-N materials simulation" J. M. Soler, E. Artacho, J. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal (to appear soon)

  Extensive description of the Siesta method.

For more information you can visit the web page `http://www.uam.es/siesta`.

The following is a short description of the compilation procedures and of the datafile format for the Siesta code.

2 QUICK START

2.1 Compilation

Unpack the SIESTA distribution. Go to the `Src/Sys` subdirectory and take a look at the `.make` files there. If one of them suits your platform, copy it or link it to `Src/arch.make`. If you intend to create a parallel version of SIESTA, make sure you have all the extra support libraries (`MPI, scalapack, blacs...`). Type `make`. The executable should work for any job (This is not exactly true, since some of the parameters in the atomic routines are still hardwired (see `Src/atmparams.f`) , but those would seldom need to be changed.)

2.2 Running the program

Some example FDF files are provided in the `Examples` directory. (Actually, this directory contains basically the `.fdf` files and the pseudopotential generation input files. A faster way to test your installation of SIESTA is implemented in directory `Src/Tests`. On the other hand, at some point you will have to generate your own pseudopotentials and run your own jobs, so what follows is also of interest.)

Probably the simplest to start with is the water-molecule example. It is advisable to create independent subdirectories for each job, so that everything is clean and neat, and out of the `siesta` directory, so that one can easily update version by replacing the whole `siesta` tree:

```
mkdir h2o
```

We need to generate the required pseudopotentials (We are going to streamline this process for this time, but you must realize that this is a tricky business that you must master before using SIESTA responsibly. Every pseudopotential must be thoroughly checked before use. Please refer to the ATOM program manual in `~/siesta/Pseudo/atom/Docs` for details regarding what follows.)

```
$ cd ~/siesta/Pseudo/atom
```

```
$ make
```

Now the pseudopotential-generation program, called atm, should be compiled.

```
$ cd Samples
```

```
$ cat o.tm2.inp
```

This is the input file, for the oxygen pseudopotential, that we have prepared for you. It is in a standard (but obscure) format that you will need to understand in the future:

```
------------------------------------------------------------
   pg       Oxygen
        tm2  2.0
 n=O  c=ca
       0.0       0.0       0.0       0.0       0.0       0.0
    1    4
    2    0     2.00      0.00
```

```
    2    1     4.00      0.00
    3    2     0.00      0.00
    4    3     0.00      0.00
   1.15     1.15     1.15     1.15
------------------------------------------------------------
```

```
$ pg.sh o.tm2.inp
```

Now there should be a new subdirectory called o.tm2 (o for o-xygen) and `o.tm2.vps` (unformatted) and `o.tm2.psf` (ASCII) files.

```
$ cp o.tm2.psf ∼/siesta/h2o/O.psf
```

copies the generated pseudopotential file to your working directory. (The unformatted and ASCII files are functionally equivalent, but the latter is more transportable and easier to look at, if you so desire.)

```
$ pg.sh h.tm2.inp
```

```
$ cp h.tm2.psf ∼/siesta/h2o/H.psf
```

repeats the process for hydrogen.

Now you are ready to run the program:

```
siesta < h2o.fdf | tee h2o.out
```

(Actually, if you are running the parallel version you should use some other invocation, such as `mpirun -np 2 siesta ...`, but we cannot go into that here.)

After a successful run of the program, you should have the following files in your directory:

- out.fdf (contains all the data used, explicit or default-ed)
- O.ion and H.ion (complete information about the basis and KB projectors)
- siesta.size (contains the memory use of some subroutines and arrays)
- h2o.XV (contains the final positions and velocities)
- h2o.DM (contains the density matrix to allow a restart)
- h2o.ANI (contains the coordinates of every MD step, in this case only one)
- h2o.FA (contains the forces on the atoms)
- h2o.EIG (contains the eigenvalues of the Kohn-Sham Hamiltonian)
- h2o.out (standard output)

The last one is the standard output of the program, that you have already seen passing on the screen. Have a look at it and refer to the output-explanation section if necessary. You may also want to look at the out.fdf file to see all the default values that siesta has chosen for you, before studying the input-explanation section and start changing them.

Now look at the other data files in `Examples` (all with and .fdf suffix) choose one and repeat the process for it.

3 PSEUDOPOTENTIAL HANDLING

The atomic pseudopotentials are stored either in binary files (with extension `.vps`) or in ASCII files (with extension `.psf`), and are read at the beginning of the execution, for each species defined in the input file. The data files must be named `*.vps` (or `*.psf`), where `*` is the label of the chemical species (see the **ChemicalSpeciesLabel** descriptor below).

These files are generated by the ATOM program (read `siesta/Pseudo/atom/README` for more complete authorship and copyright acknowledgements). It is included (with permission) in siesta/Pseudo/atom. Remember that **all pseudopotentials should be thoroughly tested** before using them. We refer you to the standard literature on pseudopotentials and to the ATOM manual `siesta/Pseudo/atom/atom.tex`.

4 ATOMIC-ORBITAL BASES IMPLEMENTED IN SIESTA

The main advantage of atomic orbitals is their efficiency (fewer orbitals needed per electron for similar precision) and their main disadvantage is the lack of systematics for optimal convergence, an issue that quantum chemists have been working on for many years. They have also clearly shown that there is no limitation on precision intrinsic to LCAO. This section provides some information about how basis sets can be generated for SIESTA.

It is important to stress at this point that neither the SIESTA method nor the program are bound to the use of any particular kind of atomic orbitals. The user can feed to SIESTA the atomic basis set he/she choses by means of radial tables (see **User.Basis** below), the only limitations being: (*i*) the functions have to be atomic-like (radial functions mutiplied by spherical harmonics), and (*ii*) they have to be of finite support, i.e., each orbital becomes strictly zero beyond some cutoff radius chosen by the user.

Most users, however, do not have their own basis sets. For these users we have devised some schemes to generate reasonable basis sets within the program. These bases depend on several parameters per atomic species that are for the user to choose, and can be important for both quality and efficiency. A description of these bases and some performance tests can be found in

"Numerical atomic orbitals for linear-scaling calculations" J. Junquera, O. Paz, D. Sánchez-Portal, and E. Artacho, Phys. Rev. B (in press, preprint in cond-mat/0104170).

An important point here is that the basis set selection is a variational problem and, therefore, minimizing the energy with respect to any parameters defining the basis is an "ab initio" way to define them.

We have also devised a quite simple and systematic way of generating basis sets based on specifying only one main parameter (the energy shift) besides the basis size. It does not offer the best NAO results one can get for a given basis size but it has the important advantages mentioned above. More about it in:

"Linear-scaling ab-initio calulations for large and complex systems" E. Artacho, D. Sánchez-Portal, P. Ordejón, A. García and J. M. Soler, Phys. Stat. Sol. (b) **215**, 809 (1999).

(Note: In addition to SIESTA we provide the program GEN-BASIS, which reads SIESTA's input and generates basis files for later use. GEN-BASIS is compiled automatically at the same time as

SIESTA. It should be run from the `Tutorials/Bases` directory, using the `gen-basis.sh` script. It is limited to a single species.

In the following we give some clues on the basics of the basis sets that SIESTA generates. The starting point is always the solution of Kohn-Sham's Hamiltonian for the isolated pseudo-atoms, solved in a radial grid, with the same approximations as for the solid or molecule (the same exchange-correlation functional and pseudopotential), plus some way of confinement (see below). We describe in the following three main features of a basis set of atomic orbitals: size, range, and radial shape.

4.1 Size: number of orbitals per atom

Following the nomenclature of Quantum Chemistry, we establish a hierarchy of basis sets, from single-ζ to multiple-ζ with polarization and diffuse orbitals, covering from quick calculations of low quality to high precision, as high as the finest obtained in Quantum Chemistry. A single-ζ (also called minimal) basis set (SZ in the following) has one single radial function per angular momentum channel, and only for those angular momenta with substantial electronic population in the valence of the free atom. It offers quick calculations and some insight on qualitative trends in the chemical bonding and other properties. It remains too rigid, however, for more quantitative calculations requiring both radial and angular flexibilization.

Starting by the radial flexibilization of SZ, a better basis is obtained by adding a second function per channel: double-ζ (DZ). In Quantum Chemistry, the *split valence* scheme is widely used: starting from the expansion in Gaussians of one atomic orbital, the most contracted gaussians are used to define the first orbital of the double-ζ and the most extended ones for the second. For strictly localized functions there was a first proposal of using the excited states of the confined atoms, but it would work only for tight confinement (see **PAO.BasisType** `nodes` below). We then proposed an extension of the split valence idea of Quantum Chemistry to strictly localized NAO which has become the standard and has been used quite succesfully in many systems (see **PAO.BasisType** `split` below). It is based on the idea of suplementing the first ζ with, instead of a gaussian, a numerical orbital that reproduces the tail of the original PAO outside a matching radius r_m, and continues smoothly towards the origin as $r^l(a - br^2)$, with a and b ensuring continuity and differenciability at r_m. Within exactly the same Hilbert space, the second orbital can be chosen to be the difference between the smooth one and the original PAO, which gives a basis orbital strictly confined within the matching radius r_m, i.e., smaller than the original PAO!

Extra parameters have thus appeared: one r_m per orbital to be doubled. The user can again introduce them by hand (see **PAO.Basis** below). Alternatively, all the r_m's can be defined at once by specifying the value of the tail of the original PAO beyond r_m, the so-called split norm. Variational optimization of this split norm performed on different systems shows a very general and stable performance for values around 15% (except for the $\sim 50\%$ for hydrogen). It generalizes to multiple-ζ trivially.

Angular flexibility is obtained by adding shells of higher angular momentum. Ways to generate these so-called polarization orbitals have been described in the literature for Gaussians. For NAOs there are two ways for SIESTA and GENBASIS to generate them: (*i*) Use atomic PAO's of higher angular momentum with suitable confinement, and (*ii*) solve the pseudoatom in the presence of an electric field and obtain the $l+1$ orbitals from the perturbation of the l orbitals by the field.

Finally, the method allows the inclusion of offsite orbitals (not centered around any specific atom). The orbitals again can be of any shape, including atomic orbitals as if an atom would be there (useful for basis-set superposition-error calculations). Bessel functions can also be added to the basis set.

4.2 Range: cutoff radii of orbitals

Strictly localized orbitals (zero beyond a cutoff radius) are used in order to obtain sparse Hamiltonian and overlap matrices for linear scaling. One cutoff radius per angular momentum channel has to be given for each species. A balanced and systematic starting point for defining all the different radii is achieved by giving one single parameter, the energy shift, i.e., the energy raise suffered by the orbital when confined. Allowing for system and physical-quantity variablity, as a rule of thumb $\Delta E_{\mathrm{PAO}} \approx 100$ meV gives typical precisions within the accuracy of current GGA functionals. The user can, nevertheless, change the cutoff radii at will.

4.3 Shape

Within the pseudopotential framework it is important to keep the consistency between the pseudopotential and the form of the pseudoatomic orbitals in the core region. The shape of the orbitals at larger radii depends on the cutoff radius (see above) and on the way the localization is enforced.

The first proposal (and quite a standard among SIESTA users) uses an infinite square-well potential. It has been widely and succesfully used for minimal bases within the ab initio tight-binding scheme of Sankey and collaborators using the FIREBALL program, but also for more flexible bases using the methodology of SIESTA. This scheme has the disadavantage, however, of generating orbitals with a discontinuous derivative at r_c. This discontinuity is more pronounced for smaller r_c's and tends to disappear for long enough values of this cutoff. It does remain, however, appreciable for sensible values of r_c for those orbitals that would be very wide in the free atom. It is surprising how small an effect such kink produces in the total energy of condensed systems. It is, on the other hand, a problem for forces and stresses, especially if they are calculated using a (coarse) finite three-dimensional grid.

Another problem of this scheme is related to its defining the basis considering the free atoms. Free atoms can present extremely extended orbitals, their extension being, besides problematic, of no practical use for the calculation in condensed systems: the electrons far away from the atom can be described by the basis functions of other atoms.

A traditional scheme to deal with this is the one based on the radial scaling of the orbitals by suitable scale factors. In addition to very basic bonding arguments, it is soundly based on restoring virial's theorem for finite bases, in the case of coulombic potentials (all-electron calculations). The use of pseudopotentials limits its applicability, allowing only for extremely small deviations from unity ($\sim 1\%$) in the scale factors obtained variationally (with the exception of hydrogen that can contract up to 25%). This possiblity is available to the user.

Another way of dealing with that problem and that of the kink at the same time is adding a soft confinement potential to the atomic Hamiltonian used to generate the basis orbitals: it smoothens the kink and contracts the orbital as suited. Two additional parameters are introduced for the purpose, which can be defined again variationally. The confining potential is flat (zero) in the core region, starts off at some internal radius r_i with all derivatives continuous and diverges at r_c ensuring the strict localization there. It is

$$V(r) = V_{\rm o} \frac{e^{-\frac{r_c - r_i}{r - r_i}}}{r_c - r} \tag{1}$$

and both r_i and $V_{\rm o}$ can be given to SIESTA together with r_c in the input (see **PAO.Basis** below).

Finally, the shape of an orbital is also changed by the ionic character of the atom. Orbitals in cations tend to shrink, and they swell in anions. Introducing a δQ in the basis-generating free-atom calculations gives orbitals better adapted to ionic situations in the condensed systems.

More information about basis sets can be found in the proposed literature.

5 COMPILING THE PROGRAM

The compilation of the program is done using a Makefile that is provided with the code. This Makefile will generate the executable for any of several architectures, with a minimum of tuning required from the user. The instructions are in directory siesta/Src/Sys, where there are also a number of `.make` files already prepared for several architectures and operating sistems. If none of these fit your needs, you will have to prepare one on your own. Note that some systems require running a program such as ranlib to index object code libraries. Once you have decided which `*.make` you want to use, you just need to copy or link it to the `Src` directory as `arch.make`. Then, just compile using the `make` utility, as usual.

6 INPUT DATA FILE

6.1 The Flexible Data Format (FDF)

The main input file, which is read as the standard input (unit 5), contains all the physical data of the system and the parameters of the simulation to be performed. This file is written in a special format called FDF, developed by Alberto García and José M. Soler. This format allows data to be given in any order, or to be omitted in favor of default values. Refer to documentation in ∼/siesta/Src/fdf for details. Here we offer a glimpse of it through the following rules:

- The FDF syntax is a 'data label' followed by its value. Values that are not specified in the datafile are assigned a default value.
- FDF labels are case insensitive, and characters - _ . in a data label are ignored. Thus, LatticeConstant and lattice_constant represent the same label.
- All text following the # character is taken as comment.

- Logical values can be specified as T, true, .true., yes, F, false, .false., no. Blank is also equivalent to true.
- Character strings should **not** be in apostrophes.
- Real values which represent a physical magnitude must be followed by its units. Look at function fdf_convfac in file ~/siesta/Src/fdf/fdf.f for the units that are currently supported.
- Complex data structures are called blocks and are placed between '%block label' and a '%endblock label' (without the quotes).
- You may 'include' other FDF files and redirect the search for a particular data label to another file. If a data label appears more than once, its first appearance is used.

These are some examples:

```
SystemName      Water molecule  # This is a comment
SystemLabel     h2o
SpinPolarized        yes
SaveRho
NumberOfAtoms         64
LatticeConstant       5.42 Ang
%block LatticeVectors
         1.000  0.000  0.000
         0.000  1.000  0.000
         0.000  0.000  1.000
%endblock LatticeVectors
KgridCutoff < BZ_sampling.fdf

# Reading the coordinates from a file
%block AtomicCoordinatesAndAtomicSpecies < coordinates.data

# Even reading more FDF information from somewhere else
%include mydefaults.fdf
```

Here follows a description of the variables that you can define in your Siesta input file, with their data types and default values.

6.2 General system descriptors

SystemName (*string*): A string of one or several words containing a descriptive name of the system (max. 150 characters).

Default value: blank

SystemLabel (*string*): A **single** word (max. 20 characters **without blanks**) containing a nickname of the system, used to name output files.

Default value: `siesta`

NumberOfAtoms (*integer*): Number of atoms in the simulation.

Default value: There is no default. You must supply this variable.

NumberOfSpecies (*integer*): Number of different atomic species in the simulation. Atoms of the same species, but with a different pseudopotential or basis set are counted as different species.

Default value: There is no default. You must supply this variable.

ChemicalSpeciesLabel (*data block*): It specifies the different chemical species that are present, assigning them a number for further identification. SIESTA recognizes the different atoms by the given atomic number.

```
%block Chemical_Species_label
   1   6   C
   2  14   Si
   3  14   Si_surface
%endblock Chemical_Species_label
```

The first number in a line is the species number, it is followed by the atomic number, and then by the desired label. This label will be used to identify corresponding files, namely, pseudopotential file, user basis file, basis output file, and local pseudopotential output file.

This construction allows you to have atoms of the same species but with different basis or pseudopotential, for example.

Negative atomic numbers are used for *ghost* atoms (see **PAO.basis**).

Use: This block is mandatory.

Default: No default.

PhononLabels (*data block*): It provides the mapping between the species number and those used by the PHONON program. Note that chemically identical elements might be assigned different labels if they are not related by symmetry.

```
%block PhononLabels
   1   A   Mg
   2   B   O
%endblock PhononLabels
```

The species number is followed by the PHONON program label and by the chemical symbol.

Use: This block is mandatory if `MD.TypeOfRun` is `Phonon`.

Default: No default.

AtomicMass (*data block*): It allows the user to introduce the atomic masses of the different species used in the calculation, useful for the dynamics with isotopes, for example. If a species index is not found within the block, the natural mass for the corresponding atomic number is assumed. If the block is absent all masses are the natural ones. One line per species with the species index (integer) and the desired mass (real). The order is not important. If there is no integer and/or no real numbers within the line, the line is disregarded.

```
%block AtomicMass
   3  21.5
   1  3.2
%endblock AtomicMass
```

Default: (Block absent or empty) Natural masses assumed. For *ghost* atoms (i.e. floating orbitals), a default of 1.d30 a.u. is assigned.

NetCharge (*real*): Specify the net charge of the system (in units of $|e|$). For charged systems, the energy converges very slowly versus cell size. For molecules or atoms, a Madelung correction term is applied to the energy to make it converge much faster with cell size (this is done only if the cell is SC, FCC or BCC). For other cells, or for periodic systems (chains, slabs or bulk), this energy correction term can not be applied, and the user is warned by the program. It is not advised to do charged systems other than atoms and molecules in SC, FCC or BCC cells, unless you know what you are doing.

Use: For example, the F^- ion would have **NetCharge** = `-1`, and the Na^+ ion would have **NetCharge** = `1`. Fractional charges can also be used.

Default value: `0.0`

6.3 Basis definition

The format for the input of the basis described in this manual has experienced important changes respect to previous versions of the program. Although old fashioned input files are still readable by SIESTA, we highly recommend the use of this new format, which allows a much more flexible input.

User.Basis (*logical*):

If true, the basis, KB projector, and other information is read from files *Atomlabel*`.ion`, where *Atomlabel* is the atomic species label specified in block *ChemicalSpeciesLabel.* These files can be generated by a previous SIESTA run or (one by one) by the standalone program GEN-BASIS. No pseudopotential files are necessary.

User.Basis.NetCDF (*logical*):

If true, the basis, KB projector, and other information is read from NetCDF files *Atomlabel*`.ion.nc`, where *Atomlabel* is the atomic label specified in block *ChemicalSpeciesLabel.* These files can be generated by a previous SIESTA run or by the standalone program GEN-BASIS. No pseudopotential files are necessary.

PAO.BasisType (*string*):

The kind of basis to be generated is chosen. All are based on finite-range pseudo-atomic orbitals [PAO's of Sankey and Niklewsky, PRB 40, 3979 (1989)] The original PAO's were described only for minimal bases. SIESTA generates extended bases (multiple-ζ, polarization, and diffuse orbitals applying different schemes of choice:

- Generalization of the PAO's: uses the excited orbitals of the finite-range pseudo-atomic problem, both for multiple-ζ and for polarization [see Sánchez-Portal, Artacho, and Soler, JPCM **8**, 3859 (1996)]. Adequate for short-range orbitals.
- Multiple-ζ in the spirit of split valence, decomposing the original PAO in several pieces of different range, either defining more (and smaller) confining radii, or introducing gaussians from known bases (Huzinaga's book).

All the remaining options give the same minimal basis. The different options and their FDF descriptors are the following:

- `split:` Split-valence scheme for multiple-zeta. The split is based on different radii.
- `splitgauss:` Same as `split` but using gaussian functions $e^{-(x/\alpha_i)^2}$. The gaussian widths α_i are read instead of the scale factors (see below). There is no cutting algorithm, so that a large enough r_c should be defined for the gaussian to have decayed sufficiently.
- `nodes:` Generalized PAO's.
- `nonodes:` The original PAO's are used, multiple-zeta is generated by changing the scale-factors, instead of using the excited orbitals.

Note that, for the `split` and `nodes` cases the whole basis can be generated by SIESTA with no further information required. SIESTA will use default values as defined in the following (**PAO.BasisSize**, **PAO.EnergyShift**, and **PAO.SplitNorm**, see below).

Default value: `split`

PAO.BasisSize (*string*): It defines usual basis sizes. It has effect only if there is no block **PAO.Basis** present.

- `SZ` or `MINIMAL`: minimal or single-ζ basis.
- `DZ`: Double zeta basis, in the scheme defined by **PAO.BasisType**.
- `SZP`: Single-zeta basis plus polarization orbitals.
- `DZP` or `STANDARD`: Like `DZ` plus polarization orbitals. Polarization orbitals are constructed from perturbation theory, and they are defined so they have the minimum angular momentum l such that there are not occupied orbitals with the same l in the valence shell of the ground-state atomic configuration. They polarize the corresponding $l-1$ shell.

Default value: `STANDARD`

PAO.BasisSizes (*data block*): Block which allows to specify a different value of the variable PAO.BasisSize for each species.

```
%block    PAO.BasisSizes
     Si      DZ
     H       DZP
     O       SZP
%endblock PAO.BasisSizes
```

PAO.EnergyShift (*real energy*): A standard for orbital-confining cutoff radii. It is the excitation energy of the PAO's due to the confinement to a finite-range. It offers a general procedure for defining the confining radii of the original (first-zeta) PAO's for all the species guaranteeing the compensation of the basis. It has only effect when the block **PAO.Basis** is not present or when the radii specified in that block are zero for the first zeta.

Use: It has to be positive.

Default value: `0.02 Ry`

PAO.SplitNorm (*real*): A standard to define default sensible radii for the split-valence type of basis. It gives the amount of norm that the second-ζ split-off piece has to carry. The split radius is defined accordingly. If multiple-ζ is used, the corresponding radii are obtained by imposing smaller fractions of the SplitNorm (1/2, 1/4 ...) value as norm carried by the higher zetas. It has only effect when the block **PAO.Basis** is not present or when the radii specified in that block are zero for zetas higher than one.

Default value: `0.15` (sensible values range between 0.05 and 0.5).

PS.lmax (*data block*): Block with the maximum angular momentum of the Kleinman-Bylander projectors, `lmxkb`. This information is optional. If the block is absent, or for a species which is not mentioned inside it, SIESTA will take `lmxkb(is) = lmxo(is) + 1`, where `lmxo(is)` is the maximum angular momentum of the basis orbitals of species `is`.

```
%block Ps.lmax
     Al_adatom   3
     H           1
     O           2
%endblock Ps.lmax
```

Default: (Block absent or empty). Maximum angular momentum of the basis orbitals plus one.

PS.KBprojectors (*data block*): This block provides information about the number of Kleinman-Bylander projectors per angular momentum, and for each species, that will used in the calculation. This block is optional. If the block is absent, or for species not mentioned in it, only one projector will be used for each angular momentum. The projectors will be constructed using the eigenfunctions of the respective pseudopotentials.

This block allows to specify the number of projector for each l, and also the reference energies of the wavefunctions used to build them. The specification of the reference energies is optional. If these energies are not given, the program will use the eigenfunctions with an increasing number of nodes (if there is not bound state with the corresponding number of nodes, the "eigenstates" are taken to be just functions which are made zero at very long distance of the nucleus). The units for the energy can be optionally specified, if not, the program will assumed that are given in Rydbergs. The data provided in this block must be consistent with those read from the block **PS.lmax**.

```
%block PS.KBprojectors
    Si  3
     2   1
    -0.9     eV
     0   2
    -0.5  -1.0d4 Hartree
     1   2
    Ga  1
     1  3
    -1.0  1.0d5 -6.0
%endblock PS.KBprojectors
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present):

```
From is = 1 to  nspecies
     read: label(is), l_shells(is)
     From lsh=1 to l_shells(is)
          read: l, nkbl(l,is)
          read: {erefKB(izeta,il,is)}, from ikb = 1 to nkbl(l,is), {units}
```

When a very high energy, higher that 1000 Ry, is specified, the default is taken instead. On the other hand, very low (negative) energies, lower than -1000 Ry, are used to indicate that the energy derivative of the last state must be used. For example, in the example given above, two projectors will be used for the *s* pseudopotential of Si. One generated using a reference energy of -0.5 Hartree, and the second one using the energy derivative of this state. For the *p* pseudopotential of Ga, three projectors will be used. The second one will be constructed from an automatically generated wavefunction with one node, and the other projectors from states at -1.0 and -6.0 Rydberg.

The analysis looking for possible *ghost* states is only performed when a single projector is used. Using several projectors some attention should be paid to the "KB cosine" (kbcos), given in the output of the program. The KB cosine gives the value of the overlap between the reference state and the projector generated from it. If these numbers are very small (< 0.01, for example) for **all** the projectors of some angular momentum, one can have problems related with the presence of ghost states.

Default: (Block absent or empty). Only one KB projector, constructed from the nodeless eigenfunction, used for each angular momentum.

PAO.Basis (*data block*): Block with data to define explicitly the basis to be used. It allows the definition by hand of all the parameters that are used to construct the atomic basis. There is no need to enter information for all the species present in the calculation. The basis for the species not mentioned in this block will be generated automatically using the parameters **PAO.BasisSize**, **PAO.BasisType**, **PAO.EnergyShift** and **PAO.SplitNorm**. Some parameters can be set to zero, or left out completely. In these cases the values will be generated from the magnitudes defined above, or from the appropriate default values. For example, the radii will be obtained from **PAO.EnergyShift** or from **PAO.SplitNorm** if they are zero; the scale factors will be put to 1 if they are zero or not given in the input. An example block for a two-species calculation (H and O) is the following (`opt` means optional):

```
%block PAO.Basis        # Define Basis set
 O    2  nodes  1.0     # Label, l_shells, type (opt), ionic_charge (opt)
  n=2 0   2             # n (opt if not using semicore levels), l, Nzeta
        3.50  3.50      #     rc(izeta=1,Nzeta)(Bohr)
        0.95  1.00      #     scaleFactor(izeta=1,Nzeta) (opt)
      1   1   P   2     # l, Nzeta, PolOrb (opt), NzetaPol (opt)
        3.50            #     rc(izeta=1,Nzeta)(Bohr)
 H    1                 # Label, l_shells, type (opt), ionic_charge (opt)
      0   2             # l, Nzeta
        5.00  0.00      #     rc(izeta=1,Nzeta)(Bohr)
%endblock PAO.Basis
```

The reading is done this way (those variables in brackets are optional, therefore they are only read if present) (See the routines in `Src/basis_specs.f` for detailed information):

```
From js = 1 to  nspecies
     read: label(is), l_shells(is), { type(is) }, { ionic_charge(is) }
     From lsh=1 to l_shells(is)
          read: { n }, l, nzls(l,is), { PolOrb(l+1) }, { NzetaPol(l+1) }
          read: rcls(izeta,il,is), from izeta = 1 to nzls(l,is)
          read: { contrf(izeta,il,is) }, from izeta = 1 to nzls(l,is)
```

And here is the variable description:

- `Label`: Species label, this label determines the species index `is` according to the block **ChemicalSpecieslabel**
- `l_shells(is)`: Number of shells of orbitals with different angular momentum for species `is`
- `type(is)`: *Optional input.* Kind of basis set generation procedure for species `is`. Same options as **PAO.BasisType**
- `ionic_charge(is)`: *Optional input.* Net charge of species `is`. This is only used for basis set generation purposes. *Default value*: `0.0` (neutral atom)

- `n`: Principal quantum number of the shell. This is an optional input for normal atoms, however it must be specified when there are *semicore* states (i.e. when states that usually are not considered to belong to the valence shell have been included in the calculation)
- `l`: Angular momentum of basis orbitals of this shell
- `nzls(l,is)`: Number of 'zetas' for angular momentum l of species `is`
- `PolOrb(l+1)`: *Optional input.* If set equal to `P`, a shell of polarization functions (with angular momentum $l+1$) will be constructed from the first-zeta orbital of angular momentum l. *Default value*: ' ' (blank = No polarization orbitals).
- `NzetaPol(l+1)`: *Optional input.* Number of 'zetas' for the polarization shell (generated automatically in a split valence fashion). Only active if `PolOrb = P`. *Default value*: `1`
- `rcls(izeta,l,is)`: Cutoff radius (Bohr) of each 'zeta' for each angular momentum `l` of species `is`
- `contrf(izeta,l,is)`: *Optional input.* Contraction factor of each 'zeta' for each angular momentum `l` of species `is`. *Default value*: `1.0`

Polarization orbitals are generated by solving the atomic problem in the presence of a polarizing electric field. The orbitals are generated applying perturbation theory to the first-zeta orbital of lower angular momentum. They have the same cutoff radius than the orbitals from which they are constructed.

There is a different possibility of generating polarization orbitals: by introducing them explicitly in the **PAO.Basis** block. It has to be remembered, however, that they sometimes correspond to unbound states of the atom, their shape depending very much on the cutoff radius, not converging by increasing it, similarly to the multiple-zeta orbitals generated with the `nodes` option. Using **PAO.EnergyShift** makes no sense, and a cut off radius different from zero must be explicitely given (the same cutoff radius as the orbitals they polarize is usually a sensible choice).

A species with atomic number = -100 will be considered by SIESTA as a constant-pseudopotential atom, *i.e.*, the basis functions generated will be spherical Bessel functions with the specified r_c. In this case, r_c has to be given, **EnergyShift** will not calculate it.

Other negative atomic numbers will be interpreted by SIESTA as *ghosts* of the corresponding positive value: the orbitals are generated and put in position as determined by the coordinates, but neither pseudopotential nor electrons are considered for that ghost atom. Useful for BSSE correction.

Use: This block is optional, except when Bessel functions are present.

Default: Basis characteristics defined by global definitions given above.

6.4 Lattice, coordinates, k-sampling

LatticeConstant (*real length*): Lattice constant. This is just to define the scale of the lattice vectors.

Default value: Minimum size to include the system (assumed to be a molecule) without intercell interactions, plus 10%.

LatticeParameters (*data block*): Crystallographic way of specifying the lattice vectors, by giving six real numbers: the three vector modules, a, b, and c, and the three angles α (angle between $\vec{b}$ and $\vec{c}$), β, and γ. The three modules are in units of **LatticeConstant**, the three angles are in degrees.

Default value:

```
1.0   1.0   1.0    90.   90.  90.
```

(see the following)

LatticeVectors (*data block*): The cell vectors are read in units of the lattice constant defined above. They are read as a matrix `CELL(ixyz,ivector)`, each vector being one line.

Default value:

```
1.0    0.0    0.0
0.0    1.0    0.0
0.0    0.0    1.0
```

If the **LatticeConstant** default is used, the default of **LatticeVectors** is still diagonal but not necessarily cubic.

AtomicCoordinatesFormat (*string*): Character string to specify the format of the atomic positions in input. These can be expressed in four forms:

- `Bohr` or `NotScaledCartesianBohr` (atomic positions are given directly in Bohr, in cartesian coordinates)
- `Ang` or `NotScaledCartesianAng` (atomic positions are given directly in Ångström, in cartesian coordinates)
- `ScaledCartesian` (atomic positions are given in cartesian coordinates, in units of the lattice constant)
- `Fractional` or `ScaledByLatticeVectors` (atomic positions are given referred to the lattice vectors)

Default value: `NotScaledCartesianBohr`

AtomCoorFormatOut (*string*): Character string to specify the format of the atomic positions in output. Same possibilities as for input (**AtomicCoordinatesFormat**).

Default value: value of **AtomicCoordinatesFormat**

AtomicCoordinatesOrigin (*data block*): Vector specifying a rigid shift to apply to the atomic coordinates, given in the same format and units as these. Notice that the atomic positions (shifted or not) need not be within the cell formed by **LatticeVectors**, since periodic boundary conditions are always assumed.

Default value:

```
0.000   0.000   0.000
```

AtomicCoordinatesAndAtomicSpecies (*data block*): The reading is done this way:

```
From ia = 1 to natoms
     read: xa(ix,ia), isa(ia)
```

where `xa(ix,ia)` is the `ix` coordinate of atom `iai`, and `isa(ia)` is the species index of atom `ia`.

SuperCell (*data block*): Integer 3x3 matrix defining a supercell in terms of the unit cell:

```
%block SuperCell
   M(1,1)  M(2,1)  M(3,1)
   M(1,2)  M(2,2)  M(3,2)
   M(1,3)  M(2,3)  M(3,3)
%endblock SuperCell
```

and the supercell is defined as $SuperCell(ix, i) = \sum_j CELL(ix, j) * M(j, i)$. Notice that the matrix indexes are inverted: each input line specifies one supercell vector.

Use: The atomic positions must be given only for the unit cell, and they are 'cloned' automatically in the rest of the supercell. The **NumberOfAtoms** given must also be that in a single unit cell. However, all values in the output are given for the entire supercell. In fact, CELL is inmediately redefined as the whole supercell and the program no longer knows the existence of an underlying unit cell. All other input (apart from NumberOfAtoms and atomic positions), including **kgridMonkhorstPack** must refer to the supercell (this is a change over the previous version). Therefore, to avoid confusions, we recommend to use **SuperCell** only to generate atomic positions, and then to copy them from the output to a new input file with all the atoms specified explicitly and with the supercell given as a normal unit cell.

Default value: No supercell (supercell equal to unit cell).

GeometryConstraints (*data block*) Fixes constraints to the change of atomic coordinates during geometry relaxation or molecular dynamics. Allowed constraints are:

- `cellside`: fixes the unit-cell side lengths to their initial values (not implemented yet).
- `cellangle`: fixes the unit-cell angles to their initial values (not implemented yet).
- `stress`: fixes the specified stresses to their initial values.
- `position`: fixes the positions of the specified atoms to their initial values.

- `center`: fixes the center (mean position, not center of mass) of a group of atoms to its initial value (not implemented yet).
- `rigid`: fixes the relative positions of a group of atoms, without restricting their displacement or rotation as a rigid unit (not implemented yet).
- `routine`: Additionally, the user may write a problem-specific routine called **constr** (with the same interface as in the example below), which inputs the atomic forces and stress tensor and outputs them orthogonalized to the constraints. For example, to maintain the relative height of atoms 1 and 2:

```
      subroutine constr( cell, na, isa, amass, xa, stress, fa )
c real*8  cell(3,3)    : input lattice vectors (Bohr)
c integer na           : input number of atoms
c integer isa(na)      : input species indexes
c real*8  amass(na)    : input atomic masses
c real*8  xa(3,na)     : input atomic cartesian coordinates (Bohr)
c real*8  stress( 3,3) : input/output stress tensor (Ry/Bohr**3)
c real*8  fa(3,na)     : input/output atomic forces (Ry/Bohr)
      integer na, isa(na)
      double precision amass(na), cell(3,3), fa(3,na),
     .                 stress(3,3), xa(3,na), fz
      fz = ( fa(3,1) + fa(3,2) ) / 2.d0
      fa(3,1) = fz
      fa(3,2) = fz
      end
```

Example: consider a diatomic molecule (atoms 1 and 2) above a surface, represented by a slab of 5 atomic layers, with 10 atoms per layer. To fix the cell height, the slab's bottom layer (last 10 atoms), the molecule's interatomic distance, its height above the surface and the relative height of the two atoms (but not its azimuthal orientation and lateral position):

```
%block GeometryConstraints
   cellside   c
   cellangle  alpha  beta  gamma
   position  from -1 to -10
   rigid  1  2
   center 1  2   0.0  0.0  1.0
   stress 4 5 6
   routine constr
%endblock GeometryConstraints
```

The first line fixes the height of the unit cell, leaving the width and depth free to change (with the appropriate type of dynamics). The second line fixes all three unit-cell angles. The third line fixes all three coordinates of atoms 1 to 10, counted backwards from the last one (you may also specify a given direction, like in center). The fourth line specifies that atoms 1 and 2 form a rigid unit. The fifth line fixes the center of the molecule (atoms

1 and 2), in the z direction (0.,0.,1.). This vector is given in cartesian coordinates and, without it, all three coordinates will be fixed (to fix a center, or a position, in the x and y directions, but not in the z direction, two lines are required, one for each direction). The sixth line specifies that the stresses 4, 5 and 6 should be fixed. The convention used for numbering stresses is that 1=xx,2=yy,3=zz, 4=yz,5=xz,6=xy. The list of atoms for a given constraint may contain several atoms (as in lines 4 and 5) *or* a range (as in the third line), but not both. But you may specify many constraints of the same type, and a total of up to 10000 lines in the block. Lines may be up to 130 characters long. Ranges of atoms in a line may contain up to 1000 atoms. All names must be in lower case.

Notice that, if you only fix the position of one atom, the rest of the system will move to reach the same relative position. In order to fix the *relative* atomic position, you may fix the center of the whole system by including a line specifying 'center' without any list or range of atoms (though possibly with a direction).

Constraints are imposed by suppressing the forces in those directions, before applying them to move the atoms. For nonlinear constraints (like 'rigid'), this does not impose the exact conservation of the constrained magnitude, unless the displacement steps are very small.

If you use the *msiesta* script to compile the program, it is convenient to write your problem-dependent routine constr.f in your working directory. *msiesta* will copy this file to the siesta source directory, before compilation. To restitute the original (void) constr.f, you simply invoke *msiesta init*.

Default value: No constraints

kgrid_cutoff (*real length*): Parameter which determines the fineness of the k-grid used for Brillouin zone sampling. It is half the length of the smallest lattice vector of the supercell required to obtain the same sampling precision with a single k point. Ref: Moreno and Soler, PRB 45, 13891 (1992).

Use: If it is zero, only the gamma point is used. The resulting k-grid is chosen according to the method of Monkhorst and Pack (diagonal), and it is displaced for even numbers of mesh divisions. This parameter is not used if **kgrid_Monkhorst_Pack** is specified.

Default value: `0.0 Bohr`

kgrid_Monkhorst_Pack (*data block*): Real-space supercell, whose reciprocal unit cell is that of the k-sampling grid, and grid displacement for each grid coordinate. Specified as an integer matrix an a real vector:

```
%block kgrid_Monkhorst_Pack
   Mk(1,1)  Mk(2,1)  Mk(3,1)   dk(1)
   Mk(1,2)  Mk(2,2)  Mk(3,2)   dk(2)
   Mk(1,3)  Mk(2,3)  Mk(3,3)   dk(3)
%endblock kgrid_Monkhorst_Pack
```

where `Mk(j,i)` are integers and `dk(i)` are usually either 0.0 or 0.5. The k-grid supercell is defined from `Mk` as in block **SuperCell** above.

Use: Used only if **SolutionMethod** = `diagon`. The k-grid supercell is compatible and unrelated (except for the default value, see below) with the **SuperCell** specifier. Both supercells are given in terms of the CELL specified by the **LatticeVectors** block. If `Mk` is the identity matrix and `dk` is zero, only the Γ point of the **unit** cell is used. Overrides **kgrid_cutoff**

Default value: Γ point of the (super)cell. (Default used only when **kgrid_cutoff** is not defined).

BandLinesScale (*string*): Specifies the scale of the k vectors given in **BandLines** below. The options are:

- `pi/a` (k-vector coordinates are given in cartesian coordinates, in units of π/a, where a is the lattice constant)
- `ReciprocalLatticeVectors` (k vectors are given in reciprocal-lattice-vector coordinates)

Default value: `pi/a`

BandLines (*data block*): Specifies the lines along which band energies are calculated (usually along high-symmetry directions). An example for an FCC lattice is:

```
%block BandLines
  1  1.000  1.000  1.000  L        # Begin at L
 20  0.000  0.000  0.000  \Gamma   # 20 points from L to gamma
 25  2.000  0.000  0.000  X        # 25 points from gamma to X
 30  2.000  2.000  2.000  \Gamma   # 30 points from X to gamma
%endblock BandLines
```

where the last column is an optional LaTex label for use in the band plot. If only given points (not lines) are required, simply specify 1 in the first column of each line. The first column of the first line must be always 1.

Use: Used only if **SolutionMethod** = `diagon`. The band k points are unrelated and compatible with any k-grid used to calculate the total energy and charge density.

Default value: No band energies calculated.

6.5 DFT, Grid, SCF

Harris_functional (*logical*): Logical variable to choose between self-consistent Kohn-Sham functional or non self-consistent Harris functional to calculate energies and forces.

- `.false.` : Fully self-consistent Kohn-Sham functional.

- `.true.` : Non self consistent Harris functional. Cheap but pretty crude for some systems. The forces are computed within the Harris functional in the first SCF step. Only implemented for LDA in the Perdew-Zunger parametrization.

  When this option is choosen, the values of DM.UseSaveDM, MaxSCFIterations, and DM.MixSCF1 are automatically set up to False, 1 and False respectively, no matter whatever other specification are in the INPUT file.

Default value: `.false.`

XC.functional (*string*): Exchange-correlation functional type. May be `LDA` (local density approximation, equivalent to `LSD`) or `GGA` (Generalized Gradient Approximation).

Use: Spin polarization is defined by SpinPolarized label for both `LDA` and `GGA`. There is no difference between `LDA` and `LSD`.

Default value: `LDA`

XC.authors (*string*): Particular parametrization of the exchange-correlation functional. Options are:

- `CA` (Ceperley-Alder) equivalent to `PZ` (Perdew-Zunger). Local density approximation. Ref: Perdew and Zunger, PRB 23, 5075 (1981)
- `PW92` (Perdew-Wang-92). Local density approximation. Ref: Perdew and Wang, PRB, 45, 13244 (1992)
- `PBE` (Perdew-Burke-Ernzerhof). Generalized gradients approximation. Ref: Perdew, Burke and Ernzerhof, PRL 77, 3865 (1996)

Use: **XC.functional** and **XC.authors** must be compatible.

Default value: `PZ`

SpinPolarized (*logical*): Logical variable to choose between spin unpolarized (`.false.`) or spin polarized (`.true.`) calculation.

Default value: `.false.`

NonCollinearSpin (*logical*): If `.true.`, non-collinear spin is described using spinor wavefunctions and (2×2) spin density matrices at every grid point. Ref: T. Oda et al, PRL, **80**, 3622 (1998). Not compatible with GGA because non-collinear density functional theory has been developped only for a local functional.

Default value: `.false.`

FixSpin (*logical*): If `.true.`, the calculation is done with a fixed value of the spin of the system, defined by variable **TotalSpin**. This option can only be used for collinear spin polarized calculations.

Default value: `.false.`

TotalSpin (*real*): Value of the imposed total spin polarization of the system (in units of the electron spin, 1/2). It is only used if **FixSpin** = `.true.`

Default value: 0.0

MeshCutoff (*real energy*): Defines the equivalent plane wave cutoff for the grid.

Default value: `50 Ry`

MaxSCFIterations (*integer*): Maximum number of SCF iterations per time step.

Default value: `50`

DM.MixingWeight (*real*): Proportion α of output Density Matrix to be used for the input Density Matrix of next SCF cycle (linear mixing): $\rho_{in}^{n+1} = \alpha\rho_{out}^{n} + (1-\alpha)\rho_{in}^{n}$.

Default value: `0.25`

DM.NumberPulay (*integer*): It controls the Pulay convergence accelerator. Pulay mixing generally accelerates convergence quite significantly, and can reach convergence in cases where linear mixing cannot. The guess for the $n+1$ iteration is constructed using the input and output matrices of the **DM.NumberPulay** former SCF cycles, in the following way: $\rho_{in}^{n+1} = \alpha\bar{\rho}_{out}^{n} + (1-\alpha)\bar{\rho}_{in}^{n}$, where $\bar{\rho}_{out}^{n}$ and $\bar{\rho}_{in}^{n}$ are constructed from the previous N =**DM.NumberPulay** cycles:

$$\bar{\rho}_{out}^{n} = \sum_{i=1}^{N} \beta_i \rho_{out}^{(n-N+i)} \quad ; \quad \bar{\rho}_{in}^{n} = \sum_{i=1}^{N} \beta_i \rho_{in}^{(n-N+i)}. \tag{2}$$

The values of β_i are obtained by minimizing the distance between $\bar{\rho}_{out}^{n}$ and $\bar{\rho}_{in}^{n}$. The value of α is given by variable **DM.MixingWeight**.

If **DM.NumberPulay** is 0 or 1, simple linear mixing is performed.

Default value: `0`

DM.PulayOnFile (*logical*): Store intermediate information of Pulay mixing in files (`.true.`) or in memory (`.false.`). Memory storage can increase considerably the memory requirements for large systems. If files are used, the filenames will be `SystemLabel`.P1 and `SystemLabel`.P2, where SystemLabel is the name associated to parameter `SystemLabel`.

Default value: `.false.`

DM.NumberKick (*integer*): Option to skip the Pulay mixing earch certain number of iterations, and use a linear mixing instead. Linear mixing is done every **DM.NumberKick** iterations, using a mixing coefficient α given by variable **DM.KickMixingWeight** (instead of the usual mixing **DM.MixingWeight**). This allows in some difficult cases to bring the SCF out of loops in which the selfconsistency is stocked. If **DM.MixingWeight**=0, no linear mix is used.

Default value: `0`

DM.KickMixingWeight (*real*): Proportion α of output Density Matrix to be used for the input Density Matrix of next SCF cycle (linear mixing): $\rho_{in}^{n+1} = \alpha\rho_{out}^{n} + (1-\alpha)\rho_{in}^{n}$, for linear mixing kicks within the Pulay mixing. This mixing is done every **DM.NumberKick** cycles.

Default value: `0.50`

DM.MixSCF1 (*logical*): Logical variable to indicate whether mixing is done in the first SCF cycle or not. Usually, mixing should not be done in the first cycle, to avoid non-idempotency in density matrix from Harris or previous steps. It can be useful, though, for restarts of selfconsistency runs.

Default value: `.false.`

DM.Tolerance (*real*): Tolerance of Density Matrix. When the maximum difference between the output and the input on each element of the DM in a SCF cycle is smaller than DM.Tolerance, the selfconsistency has been achieved.

Default value: 10^{-4}

DM.InitSpinAF (*logical*): It defines the initial spin density for a spin polarized calculation. The spin density is initially constructed with the maximum possible spin polarization for each atom in its atomic configuration. This variable defines the relative orientation of the atomic spins:

- `.false.` gives ferromagnetic order (all spins up).
- `.true.` gives antiferromagnetic order. Up and down are assigned according to order in the block **AtomicCoordinatesAndAtomicSpecies**: up for the odd atoms, down for even.

Default value: `.false.`

DM.InitSpin (*data block*): It defines the initial spin density for a spin polarized calculation atom by atom. In the block there is one line per atom to be spin-polarized, containing the atom index (integer, ordinal in the block **AtomicCoordinatesAndAtomicSpecies**) and the desired initial spin-polarization (real, positive for spin up, negative for spin down). A value larger than possible will be reduced to the maximum possible polarization, keeping its sign. Maximum polarization can also be given by introducing the symbol `+` or `-` instead of the polarization value. There is no need to include a line for every atom, only for those to be polarized. The atoms not contemplated in the block will be given non-polarized initialization. For non-collinear spin, the spin direction may be specified for each atom by the polar angles theta and phi, given as the last two arguments in degrees. If not specified, theta=0 is assumed. **NonCollinearSpin** must be `.true.` to use the spin direction.

Example:

```
%block DM.InitSpin
   5  -1.   90.   0.   # Atom index, spin, theta, phi (deg)
```

```
   3   +    45. -90.
   7   -
%endblock DM.InitSpin
```

Default value: If present but empty, all atoms are not polarized. If absent, **DM.InitSpinAF** defines the polarization.

NeglNonOverlapInt (*logical*): Logical variable to neglect or compute interactions between orbitals which do not overlap. These come from the KB projectors. Neglecting them makes the Hamiltonian more sparse, and the Order-N calculation faster. USE WITH CARE!!!

Default value: `.false.`

ExternalElectricField (*data block*): It specifies an external electric field for molecules, chains and slabs. The electric field should be orthogonal to 'bulk directions', like those parallel to a slab (bulk electric fields, like in dielectrics or ferroelectrics, are not allowed). If it is not, an error message is issued and the components of the field in bulk directions are suppressed automatically. The input is a vector in cartesian coordinates, in the specified units. Example:

```
%block ExternalElectricField
   0.000  0.000  0.500  V/Ang
%endblock ExternalElectricField
```

Default value: zero field

PolarizationGrids (*data block*): If specified, the macroscopic polarization will be calculated using the geometric Berry phase approach (R.D. King-Smith, and D. Vanderbilt, PRB **47**, 1651 (1993)). In this method the electronic contribution to the macroscopic polarization, along a given direction, is calculated using a discretized version of the formula

$$P_{e,\|} = \frac{ifq_e}{8\pi^3} \int_A d\mathbf{k}_\perp \sum_{n=1}^{M} \int_0^{|G_\||} dk_\| \langle u_{\mathbf{k}n} | \frac{\delta}{\delta k_\|} | u_{\mathbf{k}n} \rangle \tag{3}$$

where f is the occupation (2 for a non-magnetic system), q_e the electron charge, M is the number of occupied bands (the system **must** be an insulator), and $u_{\mathbf{k}n}$ are the periodic Bloch functions. $\mathbf{G}_\|$ is the shortest reciprocal vector along the chosen direction.

As it can be seen in formula (3), to compute each component of the polarization we must perform a surface integration of the result of a 1-D integral in the selected direction. The grids for the calculation along the direction of each of the three lattice vectors are specified in the block **PolarizationGrids**.

```
%block PolarizationGrids
   10   3  4      yes
    2  20  2       no
    4   4 15
%endblock PolarizationGrids
```

All three grids must be specified, therefore a 3×3 matrix of integer numbers must be given: the first row specifies the grid that will be used to calculate the polarization along the direction of the first lattice vector, the second row will be used for the calculation along the the direction of the second lattice vector, and the third row for the third lattice vector. The numbers in the diagonal of the matrix specifie the number of points to be used in the one dimensional line integrals along the different directions. The other numbers specifie the mesh used in the surface integrals. The last column specifies if the bidimensional grids are going to be diplaced from the origin or not, as in the Monkhorst-Pack algorithm (PRB **13**, 5188 (1976)). This last column is optional. If the number of point in one of the grids is zero, the calculation will not be performed for this particular direction.

For example, in the given example, for the computation in the direction of the first lattice vector, 15 points will be used for the line integrals, while a 3×4 mesh will be used for the surface integration. This last grid will be displaced from the origin, so Γ will not be included in the bidimensional integral. For the directions of the second and third lattice vectors, the number of points will be 20 and 2×2, and 15 and 4×4, respectively.

It has to be stressed that the macroscopic polarization can only be meaningfully calculated using this approach for insulators. Therefore, the presence of an energy gap is necessary, and no band can cross the Fermi level. The program performs a simple check of this condition, just by counting the electrons in the unit cell (the number must be even for a non-magnetic system, and the total spin polarization must have an integer value for spin polarized systems), however is the responsability of the user to check that the system under study is actually an insulator (for both spin components if spin polarized).

The total macroscopic polarization, given in the output of the program, is the sum of the electronic contribution (calculated as the Berry phase of the valence bands), and the ionic contribution, which is simply defined as the sum of the atomic positions within the unit cell multiply by the ionic charges ($\sum_i^{N_a} Z_i \mathbf{r}_i$). In the case of the magnetic systems, the bulk polarization for each spin component has been defined as

$$\mathbf{P}^\sigma = \mathbf{P}_e^\sigma + \frac{1}{2}\sum_i^{N_a} Z_i \mathbf{r}_i \tag{4}$$

N_a is the number of atoms in the unit cell, and $\mathbf{r}_i$ and Z_i are the positions and charges of the ions.

It is also worth noting, that the macroscopic polarization given by formula (3) is only defined modulo a "quantum" of polarization (the bulk polarization per unit cell is only well defined modulo $f q_e \mathbf{R}$, being $\mathbf{R}$ an arbitrary lattice vector). However, the experimentally observable quantities are associated to changes in the polarization induced by changes on the atomic positions (dynamical charges), strains (piezoelectric tensor), etc... The calculation of those changes, between different configurations of the solid, will be well defined as long as they are smaller than the "quantum", i.e. the perturbations are small enough to create small changes in the polarization.

Use: Only compatible with **SolutionMethod** = diagon.
Default value: Empty. No calculation performed.

GridCellSampling (*data block*): For improving grid-cutoff convergence. It specifies points within the grid cell for a symmetrization sampling: the space homogeneity (traslational invariance) is broken by the grid. This symmetry breaking is clear when moving one single atom in an otherwise empty simulation cell. The total energy oscillates with the grid periodicity when moving it around, like on an egg-box. This effect tends to disappear with finer grids. For a given grid it can be eliminated by recovering the lost symmetry: by symmetrizing the sensitive quantities. The full symmetrization implies an integration (averaging) over the grid cell. Instead, a finite sampling can be performed.

It is a sampling of rigid displacements of the system with respect to the grid. The original grid-system setup (one point of the grid at the origin) is always calculated. It is the (0,0,0) displacement. The block **GridCellSampling** gives the additional displacements wanted for the sampling. They are given relative to the grid-cell vectors, i.e., (1,1,1) would displace to the next grid point across the body diagonal, giving an equivalent grid-system situation (a useless displacement for a sampling).

Examples: Assume a cubic cell, and therefore a (smaller) cubic grid cell. If there is no block or the block is empty, then the original (0,0,0) will be used only. The block:

```
%block GridCellSampling
   0.5    0.5    0.5
%endblock GridCellSampling
```

would use the body center as a second point in the sampling. Or:

```
%block GridCellSampling
   0.5    0.5    0.0
   0.5    0.0    0.5
   0.0    0.5    0.5
%endblock GridCellSampling
```

gives an fcc kind of sampling, and

```
%block GridCellSampling
   0.5    0.0    0.0
   0.0    0.5    0.0
   0.0    0.0    0.5
   0.0    0.5    0.5
   0.5    0.0    0.5
   0.5    0.5    0.0
   0.5    0.5    0.5
%endblock GridCellSampling
```

gives again a cubic sampling with half the original side length. It is not trivial to choose a right set of displacements so as to maximize the new 'effective' cutoff. It depends on the kind of cell. It may be automatized in the future, but it is now left to the user, who introduces the displacements manually through this block.

The quantities which are symmetrized are: (*i*) energy terms that depend on the grid, (*ii*) forces, (*iii*) stress tensor, and (*iv*) electric dipole.

The symmetrization is performed at the end of every SCF cycle. The whole cycle is done for the (0,0,0) displacement, and, when the density matrix is converged, the same (now fixed) density matrix is used to obtain the desired quantities at the other displacements (the density matrix itself is *not* symmetrized as it gives a much smaller egg-box effect). The CPU time needed for each displacement in the **GridCellSampling** block is of the order of one extra SCF iteration.

Default value: Empty.

6.6 Eigenvalue problem: order-N or diagonalization

SolutionMethod (*string*): Character string to chose between diagonalization (`diagon`) or Order-N (`OrderN`) solution of the LDA Hamiltonian.

Default value: `diagon` for 100 atoms or less in the **unit** cell, `OrderN` for more than 100 atoms.

ElectronicTemperature (*real temperature or energy*): Temperature for Fermi-Dirac distribution. Useful specially for metals, and to accelerate selfconsistency in some cases.

Use: Used only if **SolutionMethod** = `diagon`

Default value: `300.0 K`

ON.functional (*string*): Choice of order-N minimization functionals:

- `Kim`: Functional of Kim, Mauri and Galli, PRB 52, 1640 (1995).
- `Ordejon-Mauri`: Functional of Ordejón et al, or Mauri et al, see PRB 51, 1456 (1995).
- `files`: Reads localized-function information from a file and chooses automatically the functional to be used.

Use: Used only if **SolutionMethod** = `ordern`

Default value: `Kim`

ON.MaxNumIter (*integer*): Maximum number of iterations in the conjugate minimization of the electronic energy, in each SCF cycle.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `1000`

ON.etol (*real*): Relative-energy tolerance in the conjugate minimization of the electronic energy. The minimization finishes if $2(E_n - E_{n-1})/(E_n + E_{n-1}) \leq$ ON.etol.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: $\texttt{10}^{-8}$

ON.eta (*real energy*): Fermi level parameter of Kim *et al.*. This should be in the energy gap, and tuned to obtain the correct number of electrons.

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `0.0 eV`

ON.RcLWF (*real legth*): Localization redius for the Localized Wave Functions (LWF's).

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `9.5 Bohr`

ON.ChemicalPotential (*logical*): Specifies whether to calculate an order-N estimate of the Chemical Potential, by the projetion method (Goedecker and Teter, PRB **51**, 9455 (1995); Stephan, Drabold and Martin, PRB **58**, 13472 (1998)). This is done by expanding the Fermi function (or density matrix) at a given temperature, by means of Chebishev polynomials, and imposing a real space truncation on the density matrix. To obtain a realistic estimate, the temperature should be small enough (typically, smaller than the energy gap), the localization range large enough (of the order of the one you would use for the Localized Wannier Functions), and the order of the polynomial expansion sufficiently large (how large depends on the temperature; typically, 50-100).

Use: Used only if **SolutionMethod** = `OrderN`

Default value: `.false.`

ON.ChemicalPotentialUse (*logical*): Specifies whether to use the calculated estimate of the Chemical Potential, instead of the parameter **ON.eta** for the order-N energy functional minimization. This is useful if you do not know the position of the Fermi level, typically in the beginning of an order-N run.

Use: Used only if **SolutionMethod** = `OrderN`. Overrides the value of **ON.eta**. Overrides the value of **ON.ChemicalPotential**, setting it to `.true.`.

Default value: `.false.`

ON.ChemicalPotentialRc (*real legth*): Defines the cutoff radius for the density matrix or Fermi operator in the calculation of the estimate of the Chemical Potential.

Use: Used only if **SolutionMethod** = `OrderN` and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `9.5 Bohr`.

ON.ChemicalPotentialTemperature (*real temperature or energy*): Defines the temperature to be used in the Fermi function expansion in the calculation of the estimate of the Chemical Potential. To have an accurate results, this temperature should be smaller than the gap of the system.

Use: Used only if **SolutionMethod** = `OrderN`, and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `0.05 Ry`.

ON.ChemicalPotentialOrder (*integer*): Order of the Chebishev expansion to calculate the estimate of the Chemical Potential.

Use: Used only if **SolutionMethod** = `OrderN`, and **ON.ChemicalPotential** or **ON.ChemicalPotentialUse** = `.true.`

Default value: `100`

6.7 Molecular dynamics and relaxations

MD.TypeOfRun (*string*): Type of Molecular Dynamics (MD) run. Several options for MD and structural optimizations are implemented. Note that some options specified in later variables (like quenching) modify the behavior of these MD options. If the system contains just one atom, `CG` is the only available dynamics option.

- `CG` (Coordinate optimization by conjugate gradients)
- `Verlet` (Standard Verlet algorithm MD)
- Nose (MD with temperature controlled by means of a Nosé thermostat)
- `ParrinelloRahman` (MD with pressure controlled by the Parrinello-Rahman method)
- `NoseParrinelloRahman` (MD with temperature controlled by means of a Nosé thermostat and pressure controlled by the Parrinello-Rahman method)
- `Anneal` (MD with annealing to a desired temperature and/or pressure (see variable MD.AnnealOption below))
- `FC` (Compute force constants matrix for phonon calculations))
- `Phonon` (Compute forces for a specified set of atomic displacements. These are choosen with the help of the program PHONON [1]) for phonon calculations)). See also `MD.ATforPhonon` block.

Default value: `Verlet` (`CG` for one atom systems)

MD.VariableCell (*logical*): If true, the lattice is relaxed together with the atomic coordinates in the conjugate gradient minimization. It allows to target hydrostatic pressures or arbitrary stress tensors. See **MD.MaxStressTol**, **MD.TargetPressure**, **MD.TargetStress**, and **MD.PreconditionVariableCell**.

Use: Used only if MD.TypeOfRun is `CG`.

Default value: `.false.`

MD.NumCGsteps (*integer*): Maximum number of conjugate gradient minimization moves (the minimization will stop if tolerance is reached before; see MD.MaxForceTol below).

Use: Used only if MD.TypeOfRun is `CG`

Default value: `0`

---

[1] PHONON is © copyright by Krzysztof Parlinski

MD.MaxCGDispl (*real length*): Maximum atomic displacements on a `CG` optimization move.

Use: Used only if MD.TypeOfRun is `CG`

Default value: `0.2 Bohr`

MD.PreconditionVariableCell (*real length*): A length to multiply to the strain components in a variable-cell conjugate gradient minimization. The strain components enter the minimization on the same footing as the coordinates. For a good CG efficiency, this length should make the scale of energy variation with strain similar to the one due to atomic displacements. It is also used for the application of the **MD.MaxCGDispl** value to the strain components.

Use: Used only if MD.TypeOfRun is `CG` and Md.VariableCell is `.true.`

Default value: `5.0 Ang`

MD.MaxForceTol (*real force*): Force tolerance in CG coordinate optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** (see **MD.MaxStressTol** for variable cell).

Use: Used only if MD.TypeOfRun is `CG`

Default value: `0.04 eV/Ang`

MD.MaxStressTol (*real pressure*): Stress tolerance in variable-cell CG optimization. Run stops if the maximum atomic force is smaller than **MD.MaxForceTol** and the maximum stress component is smaller than **MD.MaxStressTol**.

Use: Used only if MD.TypeOfRun is `CG` and Md.VariableCell is `.true.` Special consideration is needed if used with Sankey-type basis sets, since the combination of orbital kinks at the cutoff radii and the finite-grid integration originate discontinuities in the stress components, whose magnitude depends on the cutoff radii (or energy shift) and the mesh cutoff. The tolerance has to be larger than the discontinuities to avoid endless optimizations if the target stress happens to be in a discontinuity.

Default value: `1.0 GPa`

MD.InitialTimeStep (*integer*): Initial time step of the MD simulation. In the current version of SIESTA it must be 1.

Use: Used only if MD.TypeOfRun is not `CG`

Default value: `1`

MD.FinalTimeStep (*integer*): Final time step of the MD simulation.

Use: Used only if MD.TypeOfRun is not `CG`

Default value: `1`

MD.LengthTimeStep (*real time*): Length of the time step of the MD simulation.

Use: Used only if MD.TypeOfRun is not `CG`

Default value: `1.0 fs`

MD.InitialTemperature (*real temperature or energy*): Initial temperature for MD run. The atoms are assigned random velocities drawn from the Maxwell-Bolzmann distribution with the corresponding temperature. The constraint of zero center of mass velocity is imposed.

Use: Used only if **MD.TypeOfRun** = `Verlet`, `Nose`, `ParrinelloRahman`, `NoseParrinelloRahman` or `Anneal`.

Default value: `0.0 K`

MD.Quench (*logical*): Logical option to perform a power quench during the molecular dynamics. In the power quench, each velocity component is set to zero if it is opposite to the corresponding force of that component. This affects atomic velocities, or unit-cell velocities (for cell shape optimizations).

Use: Used only if **MD.TypeOfRun** = `Verlet` or `ParrinelloRahman`. It is incompatible with Nose thermostat options. The quench option allows structural relaxations of only atomic coordinates (with **MD.TypeOfRun** = `Verlet`) or atomic coordinates AND cell shape (with **MD.TypeOfRun** = `ParrinelloRahman`).

Default value: `.false.`

MD.TargetTemperature (*real temperature or energy*): Target temperature for Nose thermostat and annealing options.

Use: Used only if **MD.TypeOfRun** = `Nose`, `NoseParrinelloRahman` or `Anneal` (if **MD.AnnealOption** = `Temperature` or `TemperatureandPressure`)

Default value: `0.0 K`

MD.TargetPressure (*real pressure*): Target pressure for Parrinello-Rahman method, variable cell CG optimizations, and annealing options.

Use: Used only if MD.TypeOfRun = `ParrinelloRahman`, `NoseParrinelloRahman`, `CG` (variable cell), or `Anneal` (if MD.AnnealOption = `Pressure` or `TemperatureandPressure`)

Default value: `0.0 GPa`

MD.TargetStress (*data block*): External or target stress tensor for variable cell optimizations. Stress components are given in a line, in the order `xx, yy, zz, xy, xz, yz`. In units of **MD.TargetPressure**, but with the opposite sign. For example, a uniaxial compressive stress of 2 GPa along the 100 direction would be given by

```
MD.TargetPressure  2. GPa
%block MD.TargetStress
    -1.0  0.0  0.0  0.0  0.0  0.0
%endblock MD.TargetStress
```

Use: Used only if MD.TypeOfRun is `CG` and MD.VariableCell is `.true.`

Default value: Hydrostatic target pressure: `-1., -1., -1., 0., 0., 0.`

MD.NoseMass (*real moment of inertia*): Generalized mass of Nose variable. This determines the time scale of the Nose variable dynamics, and the coupling of the thermal bath to the physical system.

Use: Used only if **MD.TypeOfRun** = `Nose` or `NoseParrinelloRahman`

Default value: `100.0 Ry*fs**2`

MD.ParrinelloRahmanMass (*real moment of inertia*): Generalized mass of Parrinello-Rahman variable. This determines the time scale of the Parrinello-Rahman variable dynamics, and its coupling to the physical system.

Use: Used only if **MD.TypeOfRun** = `ParrinelloRahman` or `NoseParrinelloRahman`

Default value: `100.0 Ry*fs**2`

MD.AnnealOption (*string*): Type of annealing MD to perform. The target temperature or pressure are achieved by velocity and unit cell rescaling, in a given time determined by the variable **MD.TauRelax** below.

- `Temperature` (Reach a target temperature by velocity rescaling)
- `Pressure` (Reach a target pressure by scaling of the unit cell size and shape)
- `TemperatureandPressure` (Reach a target temperature and pressure by velocity rescaling and by scaling of the unit cell size and shape)

Use: Used only if **MD.TypeOfRun** = `Anneal`

Default value: `TemperatureAndPressure`

MD.TauRelax (*real time*): Relaxation time to reach target temperature and/or pressure in annealing MD.

Use: Used only if **MD.TypeOfRun** = `Anneal`

Default value: `100.0 fs`

MD.BulkModulus (*real pressure*): Estimate (may be rough) of the bulk modulus of the system. This is needed to set the rate of change of cell shape to reach target pressure in annealing MD.

Use: Used only if **MD.TypeOfRun** = `Anneal`, when **MD.AnnealOption** = `Pressure` or `TemperatureAndPressure`

Default value: `100.0 Ry/Bohr**3`

MD.FCDispl (*real length*): Displacement to use for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

Default value: `0.04 Bohr`

MD.FCfirst (*integer*): Index of first atom to displace for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

Default value: `1`

MD.FClast (*integer*): Index of last atom to displace for the computation of the force constant matrix for phonon calculations.

Use: Used only if **MD.TypeOfRun** = `FC`.

Default value: Same as **NumberOfAtoms**

MD.ATforPhonon (*data block*): List of "symmetry irreducible" atomic displacements for which to compute forces. Each line gives the fractional displacement for an atom, identified by its number in the atom list, and by a one-character code generated by the PHONON program. These codes are put in correspondence with the species labels in block *PhononLabels*).

```
%block MD.ATforPhonon
  0.002358   0.000000   0.000000  L     1
  0.000000   0.000000   0.003488  L     1
  0.002358   0.000000   0.000000  A    33
  0.000000   0.000000   0.003488  A    33
 -0.002358   0.000000   0.000000  L     1
  0.000000   0.000000  -0.003488  L     1
 -0.002358   0.000000   0.000000  A    33
  0.000000   0.000000  -0.003488  A    33
%endblock MD.ATforPhonon
```

Note: The presence of this block atomatically sets MD.TypeOfRun to `Phonon`.

Default value: None.

6.8 Parallel options

(Note: These features are not available in all distributions.)

BlockSize (*integer*): The orbitals are distributed over the processors when running in parallel using a 1-D block-cyclic algorithm. **BlockSize** is the number of consecutive orbitals which are located on a given processor before moving to the next one. Large values of this parameter lead to poor load balancing, while small values can lead to inefficient execution. The performance of the parallel code can be optimised by varying this parameter until a suitable value is found.

Use: Controls the blocksize used for distributing orbitals over processors

Default value: 8

ProcessorY (*integer*): The mesh points are divided in the Y and Z directions over the processors in a 2-D grid. **ProcessorY** specifies the dimension of the processor grid in the Y-direction and must be a factor of the total number of processors. Ideally the processors should be divided so that the number of mesh points per processor along each axis is as similar as possible.

Use: Controls the dimensions of the 2-D processor grid for mesh distribution

Default value: Variable - chosen using multiples of factors of the total number of processors

DiagMemory (*real no units*): Whether the parallel diagonalisation of a matrix is successful or not can depend on how much workspace is available to the routine when there are clusters of eigenvalues. **DiagMemory** allows the user to increase the memory available, when necessary, to achieve successful diagonalisation and is a scale factor relative to the minimum amount of memory that SCALAPACK might need. Failure due to this parameter being too small can potentially be avoid if the option **TryMemoryIncrease** is true. The larger the number of processors being used, the larger the value of **DiagMemory** that will be needed.

Use: Controls the amount of workspace available to parallel matrix diagonalisation

Default value: 2

DiagScale (*real no units*): Whether the parallel diagonalisation of a matrix is successful or not can depend on how much workspace is available to the routine when there are clusters of eigenvalues. **DiagScale** is the scale factor by which the memory available is increased in diagonalisation when **TryMemoryIncrease** is true.

Use: Controls the rate of increase of workspace available to parallel matrix diagonalisation

Default value: 1.5

TryMemoryIncrease (*logical*): If true, this option allows the program to increase the value of **DiagMemory** automatically by an a scale factor given by **DiagScale** until enough memory is obtained to be able to run a successful matrix diagonalisation in parallel.

Use: Determines whether the program is able to iteratively increase the memory available to SCALAPACK until diagonalisation is successful

Default value: true

ParallelOverK (*logical*): For the diagonalisation there is a choice in strategy about whether to parallelise over the K points or over the orbitals. K point diagonalisation is close to perfectly parallel but is only useful where the number of K points is much larger than the number of processors and therefore orbital parallelisation is generally preferred. The exception is for metals where the unit cell is small, but the number of K points to be sampled is very large. In this last case it is recommend that this option be used.

Use: Controls whether the diagonalisation is parallelised with respect to orbitals or K points - not allowed for non-co-linear spin case.

Default value: false

6.9 Efficiency options

DirectPhi (*logical*):

In the calculation of the matrix elements on the mesh this requires the value of the orbitals on the mesh points. This array represents one of the largest uses of memory within the code. If set to true this option allows the code to generate the orbital values when needed rather than storing the values. This obviously costs more computer time but will make it possible to run larger jobs where memory is the limiting factor.,

Use: Controls whether the values of the orbitals at the mesh points are stored or calculated on the fly.

Default value: false

SaveMemory (*logical*):

When calculating values that are stored in arrays whose dimensions cannot be accurately predicted ahead of time, there are two choices as to what to do when an array bound is exceeded. Firstly, the contents can be copied into buffer arrays while the dimensions are increased and then copied back (which is the default for expensive operations) or secondly, the arrays can be re-initialised and filled from scratch after re-dimensioning. The first approach is the fastest but requires larger amounts of memory, particular in **dhscf**, whereas the second uses the minimum memory at the expense of re-calculating a number of quantities.

Use: Controls whether the program uses algorithms which save memory at the expense of CPU time by not preserving the contents of arrays when re-initialising the dimensions due to bounds being exceeded.

Default value: false

6.10 Output options

LongOutput (*logical*): SIESTA can write to standard output different data sets depending on the values for output options described below. By default SIESTA will not write most of them. They can be large for large systems (coordinates, eigenvalues, forces, etc.) and, if written to standard output, they accumulate for all the steps of the dynamics. SIESTA writes the information in other files (see Output Files) in addition to the standard output, and these can be accumulative or not.

Setting **LongOutput** to `.true.` changes the default of some options, obtaining more information in the output (verbose). In particular, it redefines the defaults for the following:

- **WriteCoorStep**: `.true.`
- **WriteForces**: `.true.`
- **WriteKpoints**: `.true.`
- **WriteEigenvalues**: `.true.`

- **WriteKbands**: `.true.`
- **WriteBands**: `.true.`
- **WriteMullikenPop** 1

The specific changing of any of these options overrides the **LongOutput** setting for it.

Default value: `.false.`

WriteCoorInitial (*logical*): It determines whether the initial atomic coordinates of the simulation are dumped into the main output file. These coordinates correspond to the ones actually used in the first step, i.e., after reading (if pertinent) the *Systemlabel.*XV file. It is not affected by the **LongOutput** flag.

Default value: `.true.`

WriteCoorStep (*logical*): If `.true.` it writes the atomic coordinates at every time or relaxation step. Otherwise it does not. They are always written in the *Systemlabel.*XV file, but overriden at every step. They can be also accumulated in the *Systemlabel.*MD or *Systemlabel.*MDX files depending on **WriteMDhistory**. Unless the contrary is specified (see **WriteMDXmol**), if **WriteCoorStep** is `.false.`, the coordinates are accumulated in XMol `xyz` format in the *Systemlabel.*ANI file. For using the SIES2ARC utility for generating a CERIUS .arc animation file, WriteCoorStep should be `.true.`

Default value: `.false.` (see **LongOutput**)

WriteForces (*logical*): If `.true.` it writes the atomic forces at every time or relaxation step. Otherwise it does not. In this case, the forces of the last step can be found in the file *Systemlabel.*FA .

Default value: `.false.` (see **LongOutput**)

WriteKpoints (*logical*): If `.true.` it writes the coordinates of the $\vec{k}$ vectors used in the grid for k-sampling, into the main output file. Otherwise, it does not.

Default value: `.false.` (see **LongOutput**)

WriteEigenvalues (*logical*): If `.true.` it writes the Hamiltonian eigenvalues for the sampling $\vec{k}$ points, in the main output file. Otherwise it does not, but writes them in the file *Systemlabel.*EIG to be used by the EIG2DOS postprocessing utility (in the Util/ directory) for obtaining the density of states.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

WriteDM (*logical*): It determines whether the density matrix is output as a *Systemlabel.*DM file or not. For large systems this file can be quite big and therefore it may be necessary to turn this option off to conserve disk space.

Default value: `.true.`

WriteKbands (*logical*): If `.true.` it writes the coordinates of the $\vec{k}$ vectors defined for band plotting, into the main output file. Otherwise, it does not.

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

WriteBands (*logical*): If `.true.` it writes the Hamiltonian eigenvalues corresponding to the $\vec{k}$ vectors defined for band plotting, in the main output file. Otherwise it does not. They are, however, dumped into the file *Systemlabel.*bands to be used by postprocessing utilities for plotting the band structure. The GNUBANDS program (found in the Util/ directory) reads the *Systemlabel.*bands from standard input and dumps to standard output a file directly plotable by GNUPLOT. [2]

Use: Only if **SolutionMethod** is `diagon`.

Default value: `.false.` (see **LongOutput**)

WriteMullikenPop (*integer*): It determines the level of Mulliken population analysis printed:

- 0 = None
- 1 = atomic and orbital charges
- 2 = 1 + atomic overlap pop.
- 3 = 2 + orbital overlap pop.

The order of the orbitals in the population lists is defined by the order of atoms. For each atom, populations for PAO orbitals and double-z, triple-z, etc... derived from them are displayed first for all the angular momenta. Then, populations for perturbative polarization orbitals are written. Within a l-shell be aware that the order is not conventional, being y, z, x for p orbitals, and xy, yz, z^2, xz, and $x^2 - y^2$ for d orbitals.

Default value: `0` (see **LongOutput**)

WriteCoorXmol (*logical*): If `.true.` it originates the writing of an extra file named *SystemLabel*`.xyz` containing the final atomic coordinates in a format directly readable by XMOL.[3] Coordinates come out in Ångström independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**.

Default value: `.false.`

WriteCoorCerius (*logical*): If `.true.` it originates the writing of an extra file named *SystemLabel*`.xtl` containing the final atomic coordinates in a format directly readable by CERIUS.[4] Coordinates come out in `Fractional` format (the same as `ScaledByLatticeVectors`) independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. If negative coordinates are to be avoided, it has to be done from the start by shifting all the coordinates rigidly to have them positive, by using **AtomicCoordinatesOrigin**.

---

[2] GNUPLOT is under © copyright of GNU software

[3] XMol is under © copyright of Research Equipment Inc., dba Minnesota Supercomputer Center Inc.

[4] CERIUS is under © copyright of Molecular Simulations Inc.

Default value: `.false.`

WriteMDXmol (*logical*): If `.true.` it originates the writing of an extra file named *SystemLabel.*`ANI` containing all the atomic coordinates of the simulation in a format directly readable by XMOL for animation. Coordinates come out in Ångström independently of what specified in **AtomicCoordinatesFormat** and in **AtomCoorFormatOut**. This file is accumulative even for different runs. There is the alternative for animation by generating a .arc file for CERIUS. It is through the SIES2ARC postprocessing utility in the Util/ directory, and it requires the coordinates to be accumulated in the output file, i.e., WriteCoorStep = `.true.`

Default value: `.false.` if WriteCoorStep is `.true.` and vice-versa.

WriteMDhistory (*logical*): If `.true.` SIESTA accumulates the molecular dynamics trajectory in the following files:

- *Systemlabel.*MD : atomic coordinates and velocities (and lattice vectors and their time derivatives, if the dynamics implies variable cell). The information is stored unformatted for postprocessing with utility programs to analyze the MD trajectory.
- *Systemlabel.*MDE : shorter description of the run, with energy, temperature, etc., per time step.

These files are accumulative even for different runs.

Default value: `.false.`

WarningMinimumAtomicDistance (*physical*): Fixes a threshold interatomic distance below which a warning message is printed.

Default value: `1.0 Bohr`

AllocReportLevel (*integer*): Sets the level of the allocation report, printed in file `SystemLabel`.alloc:

- level 0 : no report at all (the default)
- level 1 : only total memory peak and where it occurred
- level 2 : detailed report printed only at normal program termination
- level 3 : detailed report printed at every new memory peak
- level 4 : print every individual (re)allocation or deallocation

Default value: `0`

6.11 Options for saving/reading information

UseSaveData (*logical*): Instructs to use as much information as possible stored from previous runs in files `SystemLabel`.XV, `SystemLabel`.DM and `SystemLabel`.LWF, where SystemLabel is the name associated to parameter `SystemLabel`.

Use: If the required files do not exist, warnings are printed but the program does not stop.

Default value: `.false.`

DM.UseSaveDM (*logical*): Instructs to read the density matrix stored in file `SystemLabel`.DM by a previous run.

Use: If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

ON.UseSaveLWF (*logical*): Instructs to read the localized wave functions stored in file `SystemLabel`.LWF by a previous run.

Use: Used only if **SolutionMethod** is `OrderN`. If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

MD.UseSaveXV (*logical*): Instructs to read the atomic positions and velocities stored in file `SystemLabel`.XV by a previous run.

Use: If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

MD.UseSaveCG (*logical*): Instructs to read the conjugate-gradient hystory information stored in file `SystemLabel`.CG by a previous run.

Use: To get actual continuation of iterrupted CG runs, use together with **MD.UseSaveXV** = `.true.` with the XV file generated in the same run as the CG file. If the required file does not exist, a warning is printed but the program does not stop. Overrides **UseSaveData**.

Default value: `.false.`

SaveHS (*logical*): Instructs to write the hamiltonian and overlap matrices, as well as other data required to generate bands and density of states, in file `SystemLabel`.HS. This file can be read by routine IOHS, which may be used by an application program in later versions.

Use: File `SystemLabel`.HS is only written, not read, by siesta.

Default value: `.false.`

SaveRho (*logical*): Instructs to write the valence pseudocharge density at the mesh used by DHSCF, in file `SystemLabel`.RHO. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.RHO is only written, not read, by siesta.

Default value: `.false.`

SaveDeltaRho (*logical*): Instructs to write $\delta\rho(\vec{r}) = \rho(\vec{r}) - \rho_{atm}(\vec{r})$, i.e., the valence pseudocharge density minus the sum of atomic valence pseudocharge densities. It is done for the mesh points used by DHSCF and it comes in file `SystemLabel`.DRHO. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.DRHO is only written, not read, by siesta.

Default value: `.false.`

SaveElectrostaticPotential (*logical*): Instructs to write the total electrostatic potential, defined as the sum of the hartree potential plus the local pseudopotential, at the mesh used by DHSCF, in file `SystemLabel`.VH. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.VH is only written, not read, by siesta.

Default value: `.false.`

SaveTotalPotential (*logical*): Instructs to write the valence total effective local potential (local pseudopotential + Hartree + Vxc), at the mesh used by DHSCF, in file `SystemLabel`.VT. This file can be read by routine IORHO, which may be used by an application program in later versions.

Use: File `SystemLabel`.VT is only written, not read, by siesta.

Default value: `.false.`

LocalDensityOfStates (*block*): Instructs to write the LDOS, integrated between two given energies, at the mesh used by DHSCF, in file `SystemLabel`.LDOS. This file can be read by routine IORHO, which may be used by an application program in later versions. The block must be a single line with the energies of the range for LDOS integration (relative to the program's zero, i.e. the same as the eigenvalues printed by the program) and their units. An example is:

```
%block LocalDensityOfStates
   -3.50  0.00   eV
%endblock LocalDensityOfStates
```

Use: The two energies of the range must be ordered, with lowest first. File `SystemLabel`.LDOS is only written, not read, by siesta.

Default value: LDOS not calculated nor written.

ProjectedDensityOfStates (*block*):

Instructs to write the Total Density Of States (Total DOS) and the Projected Density Of States (PDOS) on the basis orbitals, between two given energies, in files `SystemLabel`.DOS and `SystemLabel`.PDOS, respectively. The block must be a single line with the energies of the range for PDOS projection, (relative to the program's zero, i.e. the same as the eigenvalues printed by the program), the peak width (an energy) for broadening the eigenvalues, the number of points in the energy window, and the energy units. An example is:

```
%block ProjectedDensityOfStates
   -20.00  10.00  0.200  500  eV
%endblock ProjectedDensityOfStates
```

Use: The two energies of the range must be ordered, with lowest first. The PDOS is only implemented when the hamiltonian is solved by diagonalization with k-points. The PDOS is not parallelized yet, so an error message is issued when we try to use it in a parallel execution.

Output: The Total DOS is dumped into a file called `SystemLabel`.DOS. The format of this file is:

Energy value, Total DOS (spin up), Total DOS (spin down)

The Projected Density Of States for all the orbitals in the unit cell is dumped sequentially into a file called `SystemLabel`.PDOS. This file is more structured, so auxiliary tools can process it easily.

In all cases, the units for the DOS are (number of states/eV), and the Total DOS, $g(\epsilon)$, is normalized as follows:

$$\int_{-\infty}^{+\infty} g(\epsilon)\, d\epsilon = number\ \ of\ \ basis\ \ orbitals\ \ in\ \ unit\ \ cell \tag{5}$$

Default value: PDOS not calculated nor written.

WriteDenchar (*logical*): Instructs to write information needed by the utility program DENCHAR (by J. Junquera) to plot the valence charge density contours. The information is written in file `SystemLabel`.PLD.

Use: File `SystemLabel`.PLD is only written, not read, by siesta.

Default value: `.false.`

6.12 User-provided basis orbitals

See *User.Basis* and *User.Basis.NetCDF* descriptors.

6.13 Pseudopotentials

The pseudopotentials will be read by SIESTA from different files, one for each defined species (species defined either in block **ChemicalSpeciesLabel**). The name of the files should be:

Chemical_label.`vps` (unformatted) or *Chemical_label*.`psf` (ASCII)

where *Chemical_label* corresponds to the label defined in the **ChemicalSpeciesLabel** block.

7 OUTPUT FILES

7.1 Standard output

SIESTA writes its main output to standard output.

A brief description follows. See the example cases in the siesta/Examples directory for illustration.

The program starts writing the version of the code which is used. Then, the input FDF file is dumped into the output file as is (except for empty lines). The program does part of the reading and digesting of the data at the beginning within the `redata` subroutine. It prints some of the information it digests. It is important to note that it is only part of it, some other information being accessed by the different subroutines when they need it during the run (in the spirit of FDF input). A complete list of the input used by the code can be found at the end in the file `out.fdf`, including defaults used by the code in the run.

After that, the program reads the pseudopotentials, factorizes them into Kleinman-Bylander form, and generates (or reads) the atomic basis set to be used in the simulation. These stages are documented in the output file.

The simulation begins after that, the output showing information of the MD (or CG) steps and the SCF cycles within. Basic descriptions of the process and results are presented. The user has the option to customize it, however, by defining different options that control the printing of informations like coordinates, forces, $\vec{k}$ points, etc. Here is a list of useful options:

- **WriteCoorInitial** for writing the initial atomic coordinates,
- **WriteCoorStep** for writing the atomic coordinates at every step,
- **WriteForces** for writing the forces on the atoms at every step,
- **WriteKpoints** for writing the coordinates of the $\vec{k}$ points used for the sampling,
- **WriteEigenvalues** for writing the eigenvalues of the Hamiltonian at those $\vec{k}$ points,
- **WriteKbands** for writing the $\vec{k}$ points used to band-structure plots,
- **WriteBands** for writing the band structure at those $\vec{k}$ points,
- **WriteMullikenPop** for writing the Mulliken population analysis at different levels of detail.

Except for the first one, which is `.true.` by default, the default of SIESTA for this options is `.false.` (or 0 for the last) which means no writing. That gives the short output format.

There is a long output possibility (verbose) defined in SIESTA, which is obtained by setting **LongOutput** to `.true.` . It changes the default of the previous flags to `.true.` (to 1 for Mulliken), with the consequent appearance of the corresponding information in the output file. Of course, the explicit setting of any of these options overrides the **LongOutput** setting of it.

7.2 Used parameters

The file *out.fdf* contains all the parameters used by SIESTA in a given run, both those specified in the input fdf file and those taken by default. They are written in fdf format, so that you may reuse them as input directly. Input data blocks are copied to the out.fdf file only if you specify the *dump* option for them.

7.3 Array sizes

The file *siesta.size* contains the memory required by the large arrays of most subroutines. Generally, only problem-dependent arrays are considered, since fixed-size arrays are generally much smaller.

7.4 Basis

SIESTA (and the standalone program GEN-BASIS) always generate the files *Atomlabel*`.ion`, where *Atomlabel* is the atomic label specified in block *ChemicalSpeciesLabel*. Optionally, if the NetCDF support subsystem is compiled in, the programs generate NetCDF files *Atomlabel*`.ion.nc`. See an Appendix for information on the optional NetCDF package.

7.5 Pseudopotentials

SIESTA uses as local pseudopotential a smooth function up to the core cutoff radius (normally the potential generated by the core positive charge spread with a gaussian form). The Kleinman-Bylander pseudopotentials are generated accordingly. They appear in the `.ion` files.

7.6 Hamiltonian and overlap matrices

(file SystemLabel.HS) See the **SaveHS** data descriptor above.

7.7 Forces on the atoms

The atomic forces of the last step are stored in the file SystemLabel.FA if they are not written to the main output. See the **WriteForces** data descriptor above.

7.8 Sampling $\vec{k}$ points

The coordinates of the $\vec{k}$ points used in the sampling are stored in the file SystemLabel.KP . See the **WriteKpoints** data descriptor above.

7.9 Charge densities and potentials

(files SystemLabel.RHO, SystemLabel.DRHO, SystemLabel.VH, SystemLabel.VT) See **SaveRho**, **SaveDeltaRho**, **SaveElectrostaticPotential**, and **SaveTotalPotential** data descriptors above.

7.10 Energy bands

(file SystemLabel.bands) The format of this file is:

FermiEnergy (all energies in eV)
kmin, kmax (along the k-lines path, i.e. range of k in the band plot)
Emin, Emax (range of all eigenvalues)
NumberOfBands, NumberOfSpins (1 or 2), NumberOfkPoints
k1, ((ek(iband,ispin,1),iband=1,NumberOfBands),ispin=1,NumberOfSpins)
k2, ek
.
.
.
klast, ek
NumberOfkLines
kAtBegOfLine1, kPointLabel
kAtEndOfLine1, kPointLabel
.
.
.
kAtEndOfLastLine, kPointLabel

The GNUBANDS postprocessing utility program (found in the Util/ directory) reads the *Systemlabel.*bands for plotting. See the **BandLines** data descriptor above for more information.

7.11 Eigenvalues

The Hamiltonian eigenvalues for the sampling $\vec{k}$ points are dumped into SystemLabel.EIG in a format analogous to SystemLabel.bands, but without the kmin, kmax, emin, emax information, and without the abscissa. The EIG2DOS postprocessing utility can be then used to obtain the density of states. See the **WriteEigenvalues** descriptor above.

7.12 Coordinates in specific formats

- **XMol:**[1] See **WriteCoorXmol** data descriptor in subsection **Output options** above for obtaining a `.xyz` file with coordinates in XMol-readable format.

---

[1] XMol is under © copyright of Research Equipment Inc., dba Minnesota Supercomputer Center Inc.

- **CERIUS:**[2] See **WriteCoorCerius** data descriptor in subsection **Output options** above for obtaining a `.xtl` file with coordinates in CERIUS-readable format. See the SIES2ARC utility in Util/ directory for generating .arc files for CERIUS animation.

7.13 Dynamics history files

The trajectory of a molecular dynamics run (or a conjugate gradient minimization) can be accumulated in different files: SystemLabel.MD, SystemLabel.MDE, and SystemLabel.ANI. The first keeps the whole trajectory information, meaning positions and velocities at every time step, including lattice vectors if the cell varies. In an unformatted form. The second gives global infomation (energy, temperature, etc), and the third has the coordintes in a form suited for XMol animation. See the **WriteMDhistory** and **WriteMDXmol** data descriptors above for information. SIESTA always append new information on these files, making them accumulative even for different runs.

The `iomd` subroutine is prepared for the changing of the unformatted file SystemLabel.MD towards the ASCII formatted files SystemLabel.MDX and SystemLabel.MDC carrying the atomic and lattice trajectories. respectively.

7.14 Force Constant Matrix file

If the dynamics option is set to the calculation of the force constants (**MD.TypeOfRun**=`FC`), the force constants matrix is written in file *SystemLabel*.`FC`. The format is the following: for the displacement of each atom in each direction, the forces on each of the other atoms is writen (divided by the value of the displacement), in units of eV/Å^2. Each line has the forces in the x, y and z direction for one of the atoms.

7.15 PHONON forces file

If the dynamics option is set to the calculation of the forces for selected displacements (**MD.TypeOfRun**=`Phonon`, and/or the block `MD.ATforPhonon` exists), the forces are written in file *SystemLabel*.`PHONON`. The format is the following: Comment line, cell vectors in Å, and for each displacement: atom displaced and its coordinates plus fractional displacement, cartesian components of forces on all the atoms in units of eV/Å.

7.16 Intermediate and restart files

- **Positions and velocities:** Every time the atoms move, either during coordinate relaxation or molecular dynamics, their positions and velocities are stored to allow a restart, in file SystemLabel.XV, where SystemLabel is the value of that FDF descriptor (or siesta by default). The shape of the unit cell and its associated 'velocity' (in Parrinello-Rahman dynamics) are also stored in this file.

---

[2] CERIUS is under © copyright of Molecular Simulations Inc.

- **Conjugate-gradient history information:** Together with the SystemLabel.XV file, the information stored in the SystemLabel.CG file allows a smooth continuation of an interrupted conjugate-gradient relaxation process.
- **Localized Wave Functions:** At the end of each conjugate gradient minimization of the energy functional, the LWF's are stored on disk. These can be used as an input for the same system in a restart, or in case something goes wrong. The LWF's are stored in sparse form in file SystemLabel.LWF

  It is important to keep very good care of this file, since the first minimizations can take MANY steps. Loosing them will mean performing the whole minimization again. It is also a good practice to save it periodically during the simulation, in case a mid-run restart is necessary.
- **Density Matrix:** At the end of each SCF cycle the Density Matrix is stored disk. These can be used as an input for the same system in a restart, or in case something goes wrong. The DM is stored in sparse form in files SystemLabel.DM If the file does not exist, the initial density matrix is build from the neutral atom charges.

  It is important NOT to use a saved DM as an starting point for a run if the conjugate gradients minimization which produced the DM file was not highly converged. Otherwise, the charge density represented by it could be far from the actual charge density, and the calculation would most probably not converge.

8 PROBLEM HANDLING

8.1 Error and warning messages

`chkdim: ERROR: In` *routine* `dimension` *parameter* `=` *value*. `It must be ...` And other similar messages.

Description: Some array dimensions which change infrequently, and do not lead to much memory use, are fixed to oversized values. This message means that one of this parameters is too small and neads to be increased. However, if this occurs and your system is not very large, or unusual in some sense, you should suspect first of a mistake in the data file (incorrect atomic positions or cell dimensions, too large cutoff radii, etc).

Fix: Check again the data file. Look for previous warnings or suspicious values in the output. If you find nothing unusual, edit the specified routine and change the corresponding parameter. After running the program, look at the siesta.size file to check that the memory use is still acceptable.

8.2 Known but unsolved problems and bugs

The following are known problems of the order-N methods used:

- The convergence of the conjugate-gradient minimization of the electronic energy in the first selfconsistency step (with **SolutionMethod** = `orderN`) may be extremely slow (up to 2000 CG iterations, compared to 20 in further selfconsistency steps).

- Adjusting the **ON.eta** parameter, so that the total charge is conserved, may be notably difficult for small-gap systems.

9 PROJECTED CHANGES AND ADDITIONS

The following are major projected changes and improvements.

- Subtraction of the energy and forces of the isolated atoms placed on the uniform real-space grid, in order to partially cancel the 'energy rippling' (egg-box effect) introduced by that grid.
- Spectral analysis of DOS weighted for different orbitals or atoms (just for the diagonalization option). The information is now available only through **SaveHS**.
- Solution of the Poisson-Boltzman equation for molecules in solution, using multigrid methods.
- Implementation of the Li-Nunes-Vanderbilt order-N functional (R. W. Nunes and D. Vanderbilt, PRB **50**, 17611 (1994)).

10 REPORTING BUGS

Your assistance is essential to help improve the program. If you find any problem, please report it back to us using the `send-bug` script in the top `SIESTA` directory, or (better) the bug reporting Web interface at `http://www.uam.es/siesta` (log in as `user`, with any non-empty password). There is online documentation available on the web page itself. Please keep in mind the following guidelines:

- To be useful, bug reports should be as detailed as possible, yet concise and to the point.
- Describe the exact steps you followed to see the problem. You might want to include a copy of the fdf file you used in the calculation, details about the pseudopotentials, etc, or provide a means for us to download the information.
- Be specific. Describe what happened and how it differs from what should have happened.
- If you have any idea about how to fix the problem, by all means tell us!
- Please make sure that your bug report includes:
  - Your name and email address. This is essential for a proper followup of the problem.
  - A brief one-line synopsis of the problem.
  - The SIESTA version in which the problem was found. We can't assume that you have the very latest version, and a problem that exists in one version may not exist in another. Use the version number printed at the top of any output file (also found in file Src/version.F).

  - The platform on which the problem was found, and the operating system and compiler version.
  - The bug database is open so that SIESTA users can look at the known problems (this, by the way, is useful to avoid receiving duplicate bug reports). If for some reason you do not want your report to be seen by others, please set the `Confidential` flag to `yes`.

- The other fields in the form can be left in their default values, but if your report concerns a documentation bug or a request for new functionality, please set the `Class` field accordingly. (The default is `sw-bug` which stands for "software bug".)
- Whether using the Web form or e-mail to submit your bug report, please limit your communication to one bug report per form or message.

You will receive an acknowledgement of your report by email, and notifications of any progress in the solution. If you want to browse the bug database, simply use the Web interface and request a "query".

11 ACKNOWLEDGMENTS

We want to acknowledge the use of a small number of routines, written by other authors, in developing the siesta code. In most cases, these routines were acquired from now-forgotten routes, and the reported authorships are based on their headings. If you detect any incorrect or incomplete attribution, or suspect that other routines may be due to different authors, please let us know.

- The main nonpublic contribution, that we thank thoroughly, are a number of routines, written by **A. R. Williams** in 1985, for the solution of the radial Schrödinger and Poisson equations in the APW code of Soler and Williams (PRB **42**, 9728 (1990)). Within SIESTA, they are kept in files arw.f and periodic_table.f, and they are used for the generation of the basis orbitals and the screened pseudopotentials.
- Routine pulayx, used for the SCF mixing, was originally written by **In-Ho Lee** in 1997.
- The exchange-correlation routines contained in file xc.f were written by J.M.Soler in 1996 and 1997, in collaboration with **C. Balbás** and **J. L. Martins**. Routine pzxc (in the same file), which implements the Perdew-Zunger LDA parametrization of xc, is based on routine velect, written by **S. Froyen**.
- A number of routines (some of them modified or with name changed) from *Numerical Recipes. The Art of Scientific Computing* by **W. H. Press, S. A. Teukolsky, W. T. Veterling and B. P. Flannery** (Cambridge U.P. 1987-1992) are kept in file recipes.f
- Some standard diagonalization routines by **B. S. Garbow** are kept in files rdiag.f and cdiag.f. Other diagonalization routines from the **EISPACK** package are in file eispack.f

- The multivariate fast fourier transform in cft.f was written by **R. C. Singleton** in 1968. It is used to solve Poisson's equation.
- Subroutine iomd.f for writing MD history in files was originally written by **J. Kohanoff**.

We want to thank very specially **O. F. Sankey, D. J. Niklewski** and **D. A. Drabold** for making the FIREBALL code available to P. Ordejón. Although we no longer use the routines in that code, it was essential in the initial development of the SIESTA project, which still uses many of the algorithms developed by them.

We thank **V. Heine** for his supporting and encouraging us in this project.

The SIESTA project has been supported by Spanish DGES through project PB95-0202 and by the Fundación Ramón Areces.

12 APPENDIX: Physical unit names recognized by FDF

Magnitude	Unit name	MKS value
mass	Kg	1.E0
mass	g	1.E-3
mass	amu	1.66054E-27
length	m	1.E0
length	cm	1.E-2
length	nm	1.E-9
length	Ang	1.E-10
length	Bohr	0.529177E-10
time	s	1.E0
time	fs	1.E-15
time	ps	1.E-12
time	ns	1.E-9
energy	J	1.E0
energy	erg	1.E-7
energy	eV	1.60219E-19
energy	meV	1.60219E-22
energy	Ry	2.17991E-18
energy	mRy	2.17991E-21
energy	Hartree	4.35982E-18
energy	K	1.38066E-23
energy	kcal/mol	6.94780E-21
energy	mHartree	4.35982E-21
energy	kJ/mol	1.6606E-21
energy	Hz	6.6262E-34
energy	THz	6.6262E-22
energy	cm-1	1.986E-23
energy	cm**-1	1.986E-23
force	N	1.E0
force	eV/Ang	1.60219E-9
force	Ry/Bohr	4.11943E-8

Magnitude	Unit name	MKS value
pressure	Pa	1.E0
pressure	MPa	1.E6
pressure	GPa	1.E9
pressure	atm	1.01325E5
pressure	bar	1.E5
pressure	Mbar	1.E11
pressure	Ry/Bohr**3	1.47108E13
pressure	eV/Ang**3	1.60219E11
charge	C	1.E0
charge	e	1.602177E-19
dipole	C*m	1.E0
dipole	D	3.33564E-30
dipole	debye	3.33564E-30
dipole	e*Bohr	8.47835E-30
dipole	e*Ang	1.602177E-29
MomInert	Kg*m**2	1.E0
MomInert	Ry*fs**2	2.17991E-48
Efield	V/m	1.E0
Efield	V/nm	1.E9
Efield	V/Ang	1.E10
Efield	V/Bohr	1.8897268E10
Efield	Ry/Bohr/e	2.5711273E11
Efield	Har/Bohr/e	5.1422546E11

13 APPENDIX: NetCDF

From the NetCDF User's Guide:

> The purpose of the Network Common Data Form (netCDF) interface is to allow you to create, access, and share array-oriented data in a form that is self-describing and portable. "Self-describing" means that a dataset includes information defining the data it contains. "Portable" means that the data in a dataset is represented in a form that can be accessed by computers with different ways of storing integers, characters, and floating-point numbers. Using the netCDF interface for creating new datasets makes the data portable. Using the netCDF interface in software for data access, management, analysis, and display can make the software more generally useful.
>
> [...]
>
> NetCDF is an abstraction that supports a view of data as a collection of self-describing, portable objects that can be accessed through a simple interface. Array values may be accessed directly, without knowing details of how the data are stored. Auxiliary information about the data, such as what units are used, may be stored with the data. Generic utilities and application programs can access netCDF datasets and transform, combine, analyze, or display specified fields of the data. The development of such applications may lead to improved accessibility of data and improved reusability of software for array-oriented data management, analysis, and display.

In the context of electronic structure calculations, such an interface is useful to share pseudopotential, wavefunction, and other files among different computers, regardless of their native floating point format or their endian-ness. At present, some degree of transportability can be achieved by using ascii-binary converters. However, the other major advantage of the NetCDF format, the self-description of the data and the ease of accessibility is of great interest also.

> A netCDF dataset contains dimensions, variables, and attributes, which all have both a name and an ID number by which they are identified. These components can be used together to capture the meaning of data and relations among data fields in an array-oriented dataset. The netCDF library allows simultaneous access to multiple netCDF datasets which are identified by dataset ID numbers, in addition to ordinary file names.

To be able to generate NetCDF files in SIESTA, the public domain NetCDF library (V. 3.5 or higher) must be instaled. It can be downloaded from

`http://www.unidata.ucar.edu/packages/netcdf/index.html`.

In the `arch.make` file, the following lines must exist:

```
NETCDF_LIBS=-L/path/to/netcdf-3.5/library/directory -lnetcdf
NETCDF_INTERFACE=libnetcdf_f90.a
DEFS_CDF=-DCDF
```

While it might seem a hassle to have to install the library, the added functionality is quite large, and it is set to grow in future releases of SIESTA. The area in which this is already beginning to be felt is in the visualization of atomic information. When a NetCDF file is accessed by means of a scripting language with the proper interface (Python, `http://www.python.org` is highly recommended in this regard), one can explore and plot its contents very easily. See the Utils/PyAtom directory for some example scripts.

14 APPENDIX: Parallel SIESTA

(Note: This feature might not be available in all distributions.)

At present, SIESTA has been parallelised with moderate system sizes in mind and is suitable for comensurately moderate parallel computing systems of the type most widely available. A version suitable for massively parallel systems in order to tackle grand challenge problems will hopefully be available in the future.

Apart from the possibility of faster real time performance, there is another major driving force for the use of the parallel version. All significant parts of the code have been written using a distributed data strategy over the Nodes. This means that the use of a parallel machine can allow access to a larger amount of physical memory.

Given the targets for the present version, the strategy for parallelism does not employ spatial decomposition since this is only beneficial for very large problem sizes. Hence the work is divided in 2 ways depending on the section of the code :

- For operations that are orbital based, a 1-D block cyclic distribution has been used to divide the work over processors. This is controlled by the parameter **BlockSize**. For optimal performance, this parameter should be adjusted according to the size of problem and the machine being used. Very small and very large values tend to be inefficient and typically values in the range 8 - 20 tend to be optimal. Parts of the code that parallelise in this way are, evaluation of the kinetic energy, the non-local pseudopotential contribution, determination of the overlap integrals and matrix diagonalisation/order N.
- For operations that are grid based, a 2-D block cyclic distribution over mesh points has been used to divide the work. The mesh is divided in the Y and Z directions, but not currently in the X direction. How the mesh points are divided is controlled by the **ProcessorY** option which must be a factor of the total number of mesh points in the Y direction. Performance will be optimal when the load is balanced evenly over all processors. For dense bulk materials this is straightforward to achieve. For surfaces, where there is a region of vacuum, it is worth ensuring that the mesh is divided so as to ensure that some processors do not have just vacuum regions. Parts of the code that parallelise in this way are anything connected to the mesh (i.e. within DHSCF), including the evaluation of the Hartree and exchange-correlation energies.

There is also a second mode in which the parallel version can be used. For systems where the number of K points is very large and the size of the Hamiltonian/Overlap matrices is small, then the work can be parallelised over K points. This is far more efficient in the diagonalisation step since this phase becomes embarrassingly parallel once the matrices have been distributed to each Node. This mode is selected using the **ParallelOverK** option.

In order to use the parallel version of the code you must have the following libraries installed on your computer :

```
(a) MPI :       The Message Passing Interface library - this allows the
```

```
                processors to communicate. Most machine vendors have their
                own implementations available for their own platforms.
                However, there are two freely available versions that can
                be installed :

                MPICH :

                  http://www-unix.mcs.anl.gov/mpi/mpich/

                LAMMPI :

                  http://www.lam-mpi.org/

(b) Blacs :     This is a communications library that runs on top of MPI. Again
                if can be obtained for free from :

                  http://www.netlib.org/

                Both source code and pre-compiled binaries are available.

(c) Scalapack : This is a parallel library for dense linear algebra, equivalent
                to "lapack" but for parallel systems. Once again this is freely
                available as source code or in precompiled form from :

                  http://www.netlib.org/
```

Parallel versions of the files for `arch.make` suitable for a number of systems are provided in the `Src/Sys` directory. Should there be no suitable file there for your system, then the following are the key variables to be set in the `arch.make` file:

```
MPI_INTERFACE=libmpi_f90.a
MPI_INCLUDE=/usr/local/include
DEFS_MPI=-DMPI
#
LIBS= -lscalapack -lblacs -lmpi
```

Here `MPI_INTERFACE` indicates that the interface to MPI provided should be used which handles the issue of the variable type being passed. This will be needed in nearly all cases. `MPI_INCLUDE` indicates the directory where the header file ”mpif.h” can be found on the present machine. The environment variable `DEFS_MPI` should always be set to `"-DMPI"`, since this causes the preprocessor to include the parallel code in the source. Finally `LIBS` must now include all the libraries required - namely Scalapack, Blacs and MPI, in addition to any machine optimised Blas, etc.

To execute the parallel version, on most machine, the command will now be of the form :

```
mpirun -np <nproc> siesta < input.fdf > output
```

Where `<nproc>` is the desired number of processors, `input.fdf` is the SIESTA input file and `output` is the name of the output file.

Finally, a word concerning performance of parallel execution. This is a very variable quantity and depends on the exact system you are using since it will vary according to the latency and bandwidth of the communication mechanism. This is a function of the means by which the processors are physically connected and by software factors relating to the implementation of MPI. The one almost universal truth is that for significant system sizes is that parallel diagonalisation becomes the bottleneck and the place where efficiency is most readily lost. This is basically just the nature of diagonalisation, but it is always worth tuning the BlockSize parameter.

Index