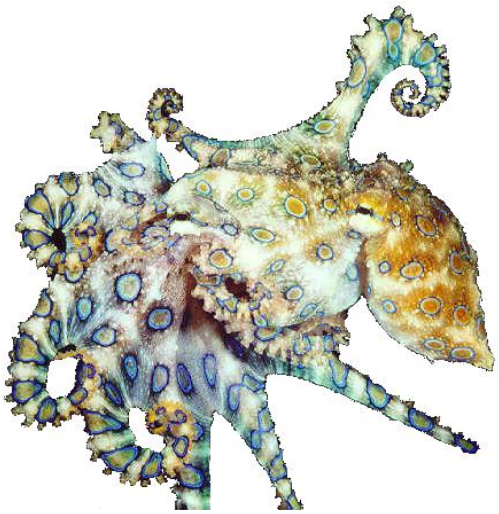
the octopus manual

Version 2.0.1

Electronic Structure
Molecular Dynamics
Excited-State Dynamics
Recipes-Generator
[No value for "UPDATED-MONTH"]



Male Hapalochlaena lunulata (top),
and female Hapalochlaena lunulata (bottom).
Photograph by Roy Caldwell.

Miguel A. L. Marques
Alberto Castro
Angel Rubio.

This manual is for octopus 2.0.1, a first principles, electronic structure, excited states, time-dependent density functional theory program.

Table of Contents

1 Authors

The main developing team of this program is composed of:

- Alberto Castro (alberto.castro@tddft.org)
- Angel Rubio (arubio@sc.ehu.es)
- Carlo Andrea Rozzi (rozzi@unimo.it)
- Florian Lorenzen (lorenzen@physik.fu-berlin.de)
- Heiko Appel (appel@physik.fu-berlin.de)
- Micael Oliveira (micael@teor.fis.uc.pt)
- Miguel A. L. Marques (marques@tddft.org)
- Xavier Andrade (andrade@mirto.polytechnique.fr)

Other contributors are:

- Sebastien Hamel - paralel version of oct-excite.

octopus is based on a fixed-nucleus code written by George F. Bertsch and K. Yabana to perform real-time dynamics in clusters (Phys Rev B **54**, 4484 (1996)) and on a condensed matter real-space plane-wave based code written by A. Rubio, X. Blase and S.G. Louie (Phys. Rev. Lett. **77**, 247 (1996)). The code was afterwards extended to handle periodic systems by G.F. Bertsch, J.I. Iwata, A. Rubio, and K. Yabana (Phys. Rev. B, **62**, 7998 (2000)). Contemporaneously there was a major rewrite of the original cluster code to handle a vast majority of finite systems. At this point the cluster code was named "tddft".

This version was consequently enhanced and beautified by A. Castro (at the time Ph.D. student of A. Rubio), originating a fairly verbose 15,000 lines of Fortran 90/77. In the year 2000, M. Marques (aka Hyllios, aka António de Faria, corsário português), joined the A. Rubio group in Valladolid as a postdoc. Having to use "tddft" for his work, and being petulant enough to think he could structure the code better than his predecessors, he started a major rewrite of the code together with A. Castro, finishing version 0.2 of "tddft." But things were still not perfect: due to their limited experience in Fortran 90, and due to the inadequacy of this language for anything beyond a HELLO WORLD program, several parts of the code were still clumsy. Also the idea of GPLing the almost 20,000 lines arose during an alcoholic evening. So after several weeks of fantic coding and after getting rid of the Numerical Recipes code that still lingered around, octopus was born.

The present released version has been completely rewritten and keeps very little relation to the old version (even input and output files) and has been enhanced with major new flags to perform various excited-state dynamics in finite and extended systems (one-dimensional periodic chains). The code will be updated frequently and new versions can be found here (http://www.tddft.org/programs/octopus).

The main features of the present version are described in detail in *octopus: a first principles tool for excited states electron-ion dynamics*, Comp. Phys. Comm. **151**, 60 (2003). Updated references as well as results obtained with octopus will be posted regularly to the octopus web page. If you find the code useful for you research we would appreciate if you give reference to this work and previous ones.

If you have some free time, and if you feel like taking a joy ride with Fortran 90, just drop us an email (octopus@tddft.org). You can also send us patches, comments, ideas, wishes, etc. They will be included in new releases of octopus.

2 Copying

The octopus Copying Conditions

This program is "free"; this means that everyone is free to use it and free to redistribute it on a free basis. What is not allowed is to try to prevent others from further sharing any version of this program that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the program, that you receive source code or else can get it if you want it, that you can change this program or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the program, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for this program. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the license are found in the General Public Licenses that accompany it.

3 Installation

Maybe somebody else installed the octopus for you. In that case, the files should be under some directory that we can call PREFIX: the executables in PREFIX/bin (e.g. if PREFIX=/usr/local, the main octopus executable is then /usr/local/bin/octopus); the documentation in info format in PREFIX/info (so that you may see it by typing info -f PREFIX/info/octopus.info if the info program is installed in your system); some sample files in PREFIX/share/octopus/samples; the pseudopotential files that octopus will need in PREFIX/share/octopus/PP, etc.

However, you may be unlucky and that is not the case. In the following we will try to help you with the, still rather unfriendly, task of compiling and installing the octopus .

3.1 Quick instructions

For the inpatients, here goes the quick-start:

```
prompt> gzip -cd octopus<-version>.tar.gz | tar xvf -
prompt> cd octopus-<version>
prompt> ./configure
prompt> make
prompt> make install
```

This will probably **not** work, so before giving up, just read the following paragraphs.

Also, rpm and deb binaries for linux are supplied on the web-page.

3.2 Long instructions

The code is written in standard Fortran 90, with some routines written in C (and in bison, if we count the input parser). To build it you will need both a C compiler (gcc works just fine), and a Fortran 90 compiler. No free-software Fortran 90 compiler is available yet, so, if you want to chew the octopus, you will have either to help the g95 (http://g95.sourceforge.net) or gfortran (http://gcc.gnu.org/fortran) projects, or use any of the available comercial compilers.

Besides the compiler, you will also need:

1. make: most computers have it installed, otherwise just grab and install the GNU make.
2. cpp: The C preprocessor is heavily used in octopus. GNU cpp is just fine, but any cpp that accepts the -C flag (preserve comments) should work just as well.
3. FFTW: We have relied on this great library to perform the Fast Fourier Transforms (FFTs). You may grab it from here (http://www.fftw.org/). You may use FFTW version 2 as well as FFTW version 3. octopus will try first to use the latter one, since it is significantly faster in some architectures.
4. LAPACK/BLAS: Required. Our politics is to rely on these two libraries as much as possible on these libraries for the linear algebra operations. If you are running Linux, there is a fair chance they are already installed in your system. The same goes to the more heavyweight machines (alphas, IBMs, SGIs, etc.). Otherwise, just grab the source from here (http://www.netlib.org).

5. GSL: Finally that someone had the nice idea of making a public scientific library! GSL still needs to grow, but it is already quite useful and impressive. `octopus` uses splines, complex numbers, special functions, etc. from GSL, so it is a must! If you don't have it already installed in your system, you can obtain GSL from here (`http://sources.redhat.com/gsl/`). You will need version 1.0 or higher.
6. MPI: If you want to run `octopus` in multi-tentacle (parallel) mode, you will need an implementation of MPI. MPICH (`http://www-unix.mcs.anl.gov/mpi/mpich/`) works just fine in our Linux boxes.

First you should obtain the code file, `octopus<-version>.tar.gz`, (this you probably have already done). The code is freely available, and can be downloaded from `http://www.tddft.org/programs/octopus`. There exists a `cvs` server, which you can browse at `http://nautilus.fis.uc.pt/cgi-bin/cvsweb.cgi/marques/octopus/`. The sources of the cvs version (in general more unstable the the *official* distribution) may be downloaded by anonymous cvs access:

```
prompt> cvs -d :pserver:anonymous@nautilus.fis.uc.pt:/server/cvsroot
        login

prompt> cvs -d :pserver:anonymous@nautilus.fis.uc.pt:/server/cvsroot
        co marques/octopus
```

Uncompress and untar it (`gzip -cd octopus<-version>.tar.gz | tar -xvf -`). In the following, `OCTOPUS-HOME` denotes the home directory of octopus, created by the `tar` command.

The `OCTOPUS-HOME` contains the following subdirectories:

- `autom4te.cache, build, CVS, debian`: contains files related to the building system or the CVS repository. May actually not be there. Not of real interest for the plain user.
- `doc`: The documentation of `octopus` in *texinfo* format.
- `liboct`: Small C library that handles the interface to GSL and the parsing of the input file. It also contains some assorted routines that we didn't want to write in boring Fortran.
- `share/PP`: Pseudopotentials. In practice now it contains the Troullier-Martins and Hartwigsen-Goedecker-Hutter pseudopotential files.
- `share/util`: Currently, the *utilities* include a couple of IBM OpenDX networks (`wf.net`), to visualize wavefunctions, densities, etc.
- `share/samples`: A couple of sample input files.
- `src`: Fortran 90 source files. Note that these have to be preprocessed before being fed to the Fortran compiler, so do not be scared by all the # directives.

Before configuring you can (should) setup a couple of options. Although the configure script tries to guess your system settings for you, we recommend that you set explicitly the default Fortran compiler and the compiler options. For example, in `bash` you would typically do:

```
export FC=abf90
export FCFLAGS="-O -YEXT_NAMES=LCS -YEXT_SFX=_"
```

if you are using the Absoft Fortran 90 compiler on a linux machine. Also, if you have some of the required libraries in some unusual directories, these directories may be placed in the variable LDFLAGS (e.g., export LDFLAGS=$LDFLAGS:/opt/lib/).

The configuration script will try to find out which compiler you are using. Unfortunately, and due to the nature of the primitive language that octopus is programmed in, the automatic test fails very often. Often it is better to set the variable FCFLAGS by hand. These are some of the options reported to work:

'intel ifc (PIV)'
```
-u -zero -fpp1 -nbs -i_dynamic -pc80 -pad -align -unroll -O3 -r8 -ip
-tpp7 -xW
```

'absoft (i386)'
```
-O3 -YEXT_NAMES=LCS -YEXT_SFX=_
```

'absoft (opteron)'
```
-O3 -mcmodel=medium -m64 -cpu:host -YEXT_NAMES=LCS -YEXT_SFX=_
```

'NAG (opteron)'
```
-colour -kind=byte -mismatch_all -abi=64 -ieee=full -O4 -Ounroll=4
```

'pgi (opteron)'
```
-fast -mcmodel=medium -O4
```

'alpha' `-align dcommons -fast -tune host -arch host -noautomatic`

'xlf (IBM)'
```
-bmaxdata:0x80000000 -qmaxmem=-1 -qsuffix=f=f90 -Q -O5 -qstrict
-qtune=auto -qarch=auto -qhot -qipa
```

'sgi' `-O3 -INLINE -n32 -LANG:recursive=on`

You can now run the configure script (./configure).[1] You can use a fair amount of options to spice octopus to your own taste. To obtain a full list just type ./configure --help. Some commonly used options include:

- --prefix=PREFIX: Change the base installation dir of octopus to PREFIX. The executable will be installed in PREFIX/bin, the libraries in PREFIX/lib and the documentation in PREFIX/info. PREFIX defaults to the home directory of the user who runs configure.[2]
- --with-fft=fftw2: Instruct the configure script to use the FFTW library, and specifically to use the FFTW version 2. You may also set --with-fft=fftw3 or even --disable-fft, although this last option is dis-encouraged.

[1] If you downloaded the cvs version, you will not find the configure script. In order to compile the development version you will first have to run the GNU autotools. This may be done by executing the autoreconf perl script (a part of the autoconf distribution). Note that you need to have working versions of the automake (1.8.5), autoconf (2.59) and libtool (1.5.6) programs (the versions we currently use are between parentheses). Note that the autoreconf script will likely fail if you have (much) older versions of the autotools.

[2] You may fine-tune further the instalation by making use of a set of standard options (--exec-prefix, --bindir, --datadir, --program-prefix, etc) that are described in the output of ./configure --help

- `--with-fft-lib=<lib>`: Instruct the `configure` script to look for the FFTW library exactly in the way that it is specified in the `<lib>` argument, i.e `--with-fft-lib='-L/opt/lib -lfftw3'`.
- `--with-blas=<lib>`: Instruct the `configure` script to look for the BLAS library in the way that it is specified in the `<lib>` argument.
- `--with-lapack=<lib>`: Instruct the `configure` script to look for the BLAS library in the way that it is specified in the `<lib>` argument.
- `--with-gsl-prefix=DIR`: Installation directory of the GSL library. The libraries are expected to be in `DIR/lib` and the include files in `DIR/include`. The value of `DIR` is usually found by issuing the command `gsl-config --prefix`. (If the GSL library is installed, the program `gsl-config` should be somewhere.)
- `--with-netcdf=<lib>`: NETCDF library. This is a recommended library, although not necessary.

In addition to these options, several other options have to be passed to build different executables of octopus (parallel, debugging version, etc) – See Section 3.3 [Different octopus executables], page 8.

Run `make`, and then `make install`. If everything went fine, you should now be able to taste `octopus`. Depending on the options passed to the configure script, some suffixes could be added to the generic name `octopus` — i.e. `octopus_cmplx` for the code compiled for complex wave-functions, `octopus_cmplx_mpi` for a parallel version of the code compiled for complex wave-function, and so on.

Depending on the value given to the `--prefix=PREFIX` given, the executables will reside in `PREFIX/bin`, the info file with the documentation in `PREFIX/info` (so that it may be viewed running `info -f PREFIX/info/octopus.info`), and the auxiliary files will be copied to `PREFIX/share/octopus`. The sample input files will be copied to `PREFIX/share/octopus/samples`.

The program has been tested in the following platforms:

- `i686*-linux-gnu`: with the Absoft (`http://www.absoft.com`), and the Intel (`http://www.intel.com/software/products/compilers/`) compiler.
- `alphae*`: both in Linux and in OSF/1 with Compaq's `fort` compiler.
- `powerpc-ibm-aix4.3.3.0`: with native `xlf90` compiler.
- `cray`: with native f90 compiler.
- `opteron`: with NAG, PGI, and ABSOFT compilers.
- `SGI`: with native compiler.

If you manage to compile/run `octopus` on a different platform or with a different compiler, please let us know so we can update the list. Patches to solve compiler issues are also welcomed.

Build the documentation in the format you prefer. Since you are reading this, you already have it in some format (but maybe not the correct version). Due to the power of `texinfo`, a series of formats are available, namely `dvi`, `html`, `pdf` and `info`. The `octopus.texi` source code of this document is in the `OCTOPUS-HOME/doc` directory.

3.3 Different octopus executables

By performing the standard install, you will get an executable called `octopus`, and a set of utility programs called `oct-`*something*. However, the code may be compiled differently in order to profit from special feautures (e.g parallel executables, etc). The following is a list of the options that have to be passed to the `configure` script in order to obtain these different executables.

- `--enable-complex`: Builds a version with complex wave-functions for the ground-state calculations (wave-functions are always complex for the evolution). This is needed when spinors are needed — e.g. noncollinear magnetism is going to be considered, or the spin-orbit coupling term will be used. The suffix `_cmplx` is appended to the `octopus` executable name.
- `--enable-single`: Builds a version that works in single precision mode, rather than in the standard double precision. It will work faster. Unfortunately we are not yet fully sure of the proper functioning of the code in this manner. The suffix `_single` is appended to the `octopus` executable name. (This is currently broken).
- `--enable-mpi`: Builds the parallel version (MPI) of `octopus`. The suffix `_mpi` is appended to the `octopus` executable name.

4 The parser

All input options should be in a file called "`inp`", in the directory `octopus` is run from. Alternatively, if this file is not found, standard input is read. For a fairly comprehensive example, just look at the file `OCTOPUS_HOME/share/samples/Na2` — if you installed the code (you did the `make install`), this file will also be in `PREFIX/share/octopus/samples/Na2`.

At the beginning of the program `liboct` reads the input file, parses it, and generates a list of variables that will be read by `octopus` (note that the input is case independent). There are two kind of variables, scalar values (strings or numbers), and blocks (that you may view as matrices). A scalar variable `var` can be defined by:

```
var = exp
```

`var` can contain any alphanumeric character plus "`_`", and `exp` can be a quote delimited string, a number (integer, real, or complex), a variable name, or a mathematical expression. In the expressions all arithmetic operators are supported ("a+b", "a-b", "a*b", "a/b"; for exponentiation the C syntax "a^b" is used), and the following functions can be used:

- `sqrt(x)`: The square root of `x`.
- `exp(x)`: The exponential of `x`.
- `log(x)` or `ln(x)`: The natural logarithm of `x`.
- `log10(x)`: Base 10 logarithm of `x`.
- `sin(x)`, `cos(x)`, `tan(x)`, `cot(x)`, `sec(x)`, `csc(x)`: The sinus, co-sinus, tangent, co-tangent, secant and co-secant of `x`.
- `asin(x)`, `acos(x)`, `atan(x)`, `acot(x)`, `asec(x)`, `acsc(x)`: The inverse (arc-) sinus, co-sinus, tangent, co-tangent, secant and co-secant of `x`.
- `sinh(x)`, `cosh(x)`, `tanh(x)`, `coth(x)`, `sech(x)`, `csch(x)`: The hyperbolic sinus, co-sinus, tangent, co-tangent, secant and co-secant of `x`.
- `asinh(x)`, `acosh(x)`, `atanh(x)`, `acoth(x)`, `asech(x)`, `acsch(x)`: The inverse hyperbolic sinus, co-sinus, tangent, co-tangent, secant and co-secant of `x`.

You can also use any of the predefined variables:

- `pi`: 3.141592653589793, what else is there to say?
- `e`: The base of the natural logarithms.
- `false` or `f` or `no`: False in all its flavors. For the curious, `false` is defined as 0.
- `true` or `t` or `yes`: The truthful companion of `false`. For the curious, `true` is defined as 1.

Blocks are defined as a collection of values, organised in row and column format. The syntax is the following:

```
%var
  exp | exp | exp | ...
  exp | exp | exp | ...
  ...
%
```

Rows in a block are separated by a newline, while columns are separated by the character "`|`" or by a tab. There may be any number of lines and any number of columns in a block. Note also that each line can have a different number of columns.

If octopus tries to read a variable that is not defined in the input file, it automatically assigns to it a default value. All variables read are output to the file "out.oct". If you are not sure of what the program is reading, just take a look at it. Everything following the character "#" until the end of the line is considered a comment and is simply cast into oblivion.

5 Description

5.1 Mesh

`octopus` uses a grid in real space to solve the Kohn-Sham equations. The grid is equally-spaced, but the spacings can be different for each Cartesian direction. The shape of the simulation region may also be tuned to suit the geometric configuration of the system.

5.2 Run Mode SCF

The ground-state electronic density in a Kohn-Sham (KS) based electronic structure code such as `octopus` is obtained after a self-consistent process that attempts to solve the KS equations. In essence, the problem is the following: at a given iteration step, one departs from an approximated solution – some KS eigenfunctions ψ_j^{inp}, eigenvalues ϵ_j^{inp} and density ρ^{inp}, which determines a KS "input" Hamiltonian. By diagonalizing this Hamiltonian, one obtains the corresponding "output" eigenfunctions, eigenvalues, and density. This cycle is considered to be closed, and the solution achieved, when some convergence criterion is fulfilled. In our case, we have allowed for four different criteria, to be defined below. The self consistent procedure will stop when the first of the convergence criteria is fulfilled, or either when a a maximum number of iterations has been performed.

5.3 Run Mode Unoccupied States

These variables are only used in `CalculationMode == unocc, unocc_start`. The purpose of these run modes is to calculate higher lying Kohn-Sham orbitals. For that purpose, it reads the restart information from a converged previous ground-state calculation, and builds the corresponding Hamiltonian. Then, it calculates the eigenvalues and eigenfunctions that are requested by the user.

Note: These variables are also used in `CalculationMode = td` or `td_start` if `TDOccupationalAnalysis == 1`).

5.4 Run Mode Time Dependent

When `CalculationMode = (td | td_start)`, the code performs the time propagation of the electronic orbitals and – if required – of the ionic positions. This latter task does not pose major algorithmical problems (the usual Verlet algorithms deal with that task); however the best way to propagate a Schrödinger-like equation is still unclear. Due to this fact, we provide with a rather excessive selection of possibilities for that purpose. Before describing the set of variables necessary to specify the way in which the time evolution is to be performed, it is worth making a brief introduction to the problem.

We are concerned with a set of Schrödinger-like equations for the electronic orbitals:

$$i\frac{\partial \psi_i}{\partial t} = H(t)\psi_i(t)\,,$$
$$\psi_i(t=0) = \psi_i^0\,.$$

Being the equation linear, one may formally define a linear "evolution" operator, which trasforms the initial vector into the solution at time T:

$$\psi_i(T) = U(T,0)\psi_i^0$$

Moreover, there is a formal exact expression for the evolution operator:

$$\psi_i(T) = T\exp\{-i\int_0^T d\tau H(\tau)\}\psi_i^0\,.$$

where $T\exp$ is the time-ordered exponential. If the Hamiltonian conmutes with itself at different times, we can drop the time-ordering product, and leave a simple exponential. If the Hamiltonian is time-independent – which makes it trivially self commuting, the solution is then simply written as:

$$\psi_i(T) = \exp\{-iTH\}\psi_i^0\,.$$

Unfortunately, this is not the case in general. We have to find an algorithm able to cope with time-dependent Hamiltonians, such as the self-consistent time-dependent Kohn-Sham operator, which is built "self consistently" from the varying electronic density.

The first step is to perform a time-discretization: the full propagation between 0 and T is decomposed as:

$$U(T,0) = \prod_{i=0}^{N-1} U(t_i+\delta t, t_i)\,,$$

where $t_0 = 0$, $t_N = T$, $\delta t = T/N$. So at each time step we are dealing with the problem of performing the short-time propagation:

$$\psi_i(t+\delta t) = U(t+\delta t, t)\psi_i(t) = T\exp\{-i\int_t^{t+\delta t} d\tau\, H(\tau)\}\psi(t)\,.$$

In this way, one can monitor the evolution in the interior of $[0,t]$. In fact, the possibility of monitoring the evolution is generally a requirement; this requirement imposes a natural restriction on the maximum size of δt: if $\omega_{\max}$ is the maximum frequency that we want to discern, δt should be no larger than $\approx 1/\omega_{\max}$. Below this $\delta t_{\max}$, we are free to choose δt considering performance reasons: Technically, the reason for the discretization is twofold: the time-dependence of H is alleviated, and the norm of the exponential argument is reduced (the norm increases linearly with δt).

Since we cannot drop the time-ordering product, the desired algorithm cannot be reduced, in principle, to the calculation of the action of the exponential of an operator over the initial vector. Some algorithms tailored to approximate the evolution operator, in fact, do not even require to peform such operator exponentials. Most of them, however, do rely on the calculation of one or more exponentials, such as the ones used by `octopus`. This is why in principle we need to specify two different issues: the "evolution method", and the "exponential method". In other words: we need an algorithm to approximate the evolution operator $U(t+\delta t, t)$ – which will be specified by variable `TDEvolutionMethod` – and, if this algorithm requires it, we will also need an algorithm to approximate the exponential of a matrix operator $\exp\{A\}$ – which will be specified by variable `TDExponentialMethod`.

5.5 Function output for visualization

Every given number of time iterations, or after ground-state calculations, some of the functions that characterise the system may be written to disk so that they may be analized. Files are written within "static" output directory after the self-consistent field, or within

"td.x" directories, during evolution, where "x" stands for the iteration number at which each write is done. Note that if you wish to plot any function (`OutputKSPotential = yes`, etc.), at least one of the output formats should be enabled (`OutputPlaneX = yes`, `OutputDX = yes`, `OutputNETCDF = yes`, etc.). [This is not necessary if you wish to plot the geometry (`OutputGeometry = yes`)]. Note further that the data written by `OutputAxisX`, `OutputPlaneX` etc. has always the (side) length of the longest axis; this is independent from the chosen geometry. Data points which are inexistent in the actual geometry have the value zero in those files.

5.6 Spectrum calculations

Once `octopus` has been run, results must be analyzed somehow. The most common thing is to Fourier-transform something to calculate spectra. This may be done through some utilities (`strength-function`, `hs-from-mult`, `hs-from-acc` which are described in section "External utilities." Common options read by these utilities are:

6 Input Variables

octopus has quite a few options, that we will subdivide in different groups. After the name of the option, its type and default value (when applicable) are given in parenthesis.

6.1 Generalities

- **CalculationMode**
  Section: Generalities
  Type: integer
  Default: gs

  Decides what kind of calculation is to be performed
  Options:
  - **gs** (01): Calculation of the ground state
  - **unocc** (02): Calculation of unoccupied/virtual KS states
  - **td** (03): Time-dependent calculation
  - **pol** (04): Calculation of the static polarizability
  - **geom** (05): Optimization of the geometry
  - **phonons** (06): Calculation of the vibrational modes
  - **opt_control** (07): Optimal control.
  - **pol_lr** (08): Linear-response calculation of the polarizability
  - **casida** (09): Excitations via linear-response TDDFT
  - **wave_matching** (10): Wave-matching a la Heiko
  - **bo** (98): Born-Oppenheimer-like Molecular Dynamics
  - **recipe** (99): Prints out a tasty recipe
- **Dimensions**
  Section: Generalities
  Type: integer
  Default: 3

  octopus can run in 1, 2 or 3 dimensions, depending on the value of this variable. Note that not all input variables may be available in all cases.
  Options:
  - **1**: The system is 1-dimensional
  - **2**: The system is 2-dimensional
  - **3**: The system is 3-dimensional
- **TmpDir**
  Section: Generalities
  Type: string
  Default: "tmp/"

  The name of the directory where octopus stores binary information like the wavefunctions.

6.1.1 Debug

- **DebugLevel**
  Section: Generalities::Debug
  Type: integer
  Default: 1

  This variable decides wether or not to enter debug-mode. In debugging mode, the program prints to standard error when it enters and exits the subroutines, what is the memory it is using (only, for the moment being, in Linux systems), and some other information. Useful for developers, and mandatory if you want to send a bug report to the developers and being considered. You have two options: (i) setting it to zero – or less than zero, in which case you do not run in debugging mode (this is the default), or (ii) setting it to a positive number. In this case the entries and exits to nested subroutines are only printed down to the level that is given in this variable.
- **DevelVersion**
  Section: Generalities::Debug
  Type: logical
  Default: no

  If true, allows the use of certains parts of the code that are still under development. This should not be used for production runs.
- **MPIDebugHook**
  Section: Generalities::Debug
  Type: logical
  Default: no

  When debugging the code in parallel it is usually difficult to find the origin of race conditions that appear in MPI communications. This variable introduces a facility to control separate MPI processes. If set to yes, all nodes will start up, but will get trapped in an endless loop. In every cycle of the loop each node is sleeping for one second and is then checking if a file with the name node_hook.xxx (where xxx denotes the node number) exists. A given node can only be released from the loop if the corresponding file is created. This allows to selectively run eg. a compute node first followed by the master node. Or, by reversing the file creation of the node hooks, to run the master first followed by a compute node.
- **ProfilingMode**
  Section: Generalities::Debug
  Type: logical
  Default: no

  Use this variable to run octopus in profiling mode. In this mode octopus records time spent in certain areas of the code and the number of times this code is executed. These numbers are written in './profiling.NNN/profiling.nnn' with nnn being the node number (000 in serial) and NNN the number of processors. This is mainly for development purposes. Note, however, that octopus should be compiled with –disable-debug to do proper profiling.

6.1.2 IO

- **FlushMessages**
  Section: Generalities::IO
  Type: logical
  Default: no

  In addition to writing to stdout and stderr, the code messages may also be flushed to "messages.stdout" and "messages.stderr", if this variable is set to yes.

- **RestartFileFormat**
  Section: Generalities::IO
  Type: integer
  Default: restart_plain

  Determines in which format the restart file should be written

  Options:

  - **restart_plain** (1): Binary (platform dependent) format
  - **restart_netcdf** (2): NetCDF (platform independent) format. This requires the NETCDF library.

- **WorkDir**
  Section: Generalities::IO
  Type: string
  Default: "."

  By default, all files are written and read from the working directory, i.e. the directory from which the executable was launched. This behavior can be changed by setting this variable: if you give it a name (other than ".") the files are written and read in that directory.

- **stderr**
  Section: Generalities::IO
  Type: string
  Default: "-"

  The standard error by default goes to, well, to standard error. This can be changed by setting this variable: if you give it a name (other than "-") the output stream is printed in that file instead.

- **stdout**
  Section: Generalities::IO
  Type: string
  Default: "-"

  The standard output by default goes to, well, to standard output. This can be changed by setting this variable: if you give it a name (other than "-") the output stream is printed in that file instead.

6.1.3 Parallel

- **ParallelizationGroupRanks**
  Section: Generalities::Parallel
  Type: block

  Specifies the size of the groups used for the parellization. For example (n_d, n_s, n_k) means we have n_p*n_s*n_k processors and that the k-points should be divided in n_k groups, the states in n_s groups, and each state in n_d domains.

- **ParallelizationStrategy**
  Section: Generalities::Parallel
  Type: flag
  Default: par_domains + par_states + par_kpoints

  Specifies what kind of parallelization strategy octopus should use. The values can be combined, for example "par_domains + par_states" means a combined paralellization in domains and states

  Options:

  - **serial**: Octopus will run in serial.
  - **par_domains** (1): Octopus will run parallel in domains.
  - **par_states** (2): Octopus will run parallel in states.
  - **par_kpoints** (4): Octopus will run parallel in k-points/spin.
  - **par_other** (8): Run-mode dependent. For example, in casida it means parallelization in e-h pairs

6.1.4 Units

- **Units**
  Section: Generalities::Units
  Type: string
  Default: "a.u"

  Atomic units seem to be the preferred system in the atomic and molecular physics community (despite the opinion of some of the authors of this program). Internally, the code works in atomic units. However, for input or output, some people like using eV for energies and AA for lengths. This other system of units can also be used.

  Options:

  - **"a.u"**: Atomic units
  - **"eVA"**: Electron-volts for energy, Angstrom for length.

- **UnitsInput**
  Section: Generalities::Units
  Type: string
  Default: "a.u"

  Same as "Units", but only refers to the values in the input files. That is, if UnitsInput

= "eVA", all physical values in the input files will be considered to be in eV and Angstroms.

- **UnitsOutput**
  Section: Generalities::Units
  Type: string
  Default: "a.u"

  Same as "Units", but only refers to the values in the output files. That is, if UnitsInput = "eVA", all physical values in the output files will be considered to be in eV and Angstroms.

6.2 Geometry Optimization

- **GOMaxIter**
  Section: Geometry Optimization
  Type: integer
  Default: 200

  Even if previous convergence criterium is not satisfied, minimization will stop after this number of iterations.

- **GOMethod**
  Section: Geometry Optimization
  Type: integer
  Default: steep

  Method by which the minimization is performed.

  Options:

  - **steep** (1): simple steepest descent.

- **GOStep**
  Section: Geometry Optimization
  Type: float
  Default: 0.5

  Initial step for the geometry optimizer.

- **GOTolerance**
  Section: Geometry Optimization
  Type: float
  Default: 0.0001 a.u.

  Convergence criterium to stop the minimization. In units of force; minimization is stopped when all forces on ions are smaller.

6.3 Hamiltonian

- **AtomsMagnetDirection**
  Section: Hamiltonian

Type: block

This option is only used when `GuessMagnetDensity` is set to `user_defined`. It provides a direction for each atoms magnetization vector when building the guess density. In order to do that the user should specify the coordinates of a vector that has the desired direction. The norm of the vector is ignored. Note that it is necessaty to maintain the ordering in which the species were defined in the coordinates specifications.

For spin-polarized calculations the vectors should have only one component and for non-collinear spin calculations they should have three components.

- **ClassicPotential**
  Section: Hamiltonian
  Type: integer

  If `true`, add to the external potential the potential generated by the point charges read from the PDB input (see `PBDCoordinates`).

- **GyromagneticRatio**
  Section: Hamiltonian
  Type: float
  Default: 2.0023193043768

  The gyromagnetic ratio of the electron. This is of course a physical constant, and the default value is the exact one that you should not touch, unless :

  (i) You want to disconnect the anomalous Zeeman term in the Hamiltonian (then set it to zero, this number only affects this term);

  (ii) You are using an effective Hamiltonian, as it is the case when you calculate a 2D electron gas, in which case you have an effective gyromagnetic factor that dependes on the material.

- **MultigridLevels**
  Section: Hamiltonian
  Type: integer

  Number of levels in the grid hierachy used for multigrid. Positive numbers indicate an absolute numbers of levels, negative numbers are substracted to maximum number of levels posible for the grid been used. Default is the maximum number of levels for the grid.

  Options:

  - **max_levels**: Calculate the optimous number of levels for the grid.

- **NonInteractingElectrons**
  Section: Hamiltonian
  Type: logical
  Default: no

  Sometimes it may be helpful to treat the electrons as non-interacting particles, i.e., not to take into account Hartree and exchange-correlation effects between the electrons. This variable may be used to toogle this behavior on and off

Options:

- **no**: Electrons are treated as *interacting* particles
- **yes** (1): Electrons are handled as *non-interacting* paticles

- **RelativisticCorrection**
Section: Hamiltonian
Type: integer
Default: non_relativistic

The default value means that *no* relativistic correction is used. To include spin-orbit coupling turn `RelativisticCorrection` to `spin_orbit` (this will only work when using an executable compiled for complex wave-functions, and if `SpinComponents` has been set to `non_collinear`, which ensures the use of spinors).

Options:

- **non_relativistic**: No relativistic corrections.
- **spin_orbit** (1): Spin-Orbit.
- **app_zora** (2): Approximated ZORA (Not implemented)
- **zora** (3): ZORA (Not implemented)

- **StaticElectricField**
Section: Hamiltonian
Type: block

A static constant electrical field may be added to the usual Hamiltonian, by setting the block StaticElectricField. Atomic units will be assumed always for its magnitude, regardless of the unit system specified. The three possible components of the block (which should only have one line) are the three components of the electrical field vector.

- **StaticMagneticField**
Section: Hamiltonian
Type: block

A static constant magnetic field may be added to the usual Hamiltonian, by setting the block StaticMagneticField. Atomic units will be assumed always for its magnitude, regardless of the unit system specified. The three possible components of the block (which should only have one line) are the three components of the magnetic field vector. Note that if you are running the code in 1D mode this will not work, and if you are running the code in 2D mode the magnetic field will have to be in the z-direction, so that the first two columns should be zero.

Important note: The static magnetic field may only be applied if you are using the executable compiled for real-only wavefunctions. If you get this as an error message, please remove the 'StaticMagneticField' block from you input file, or else try running with a complex executable (octopus_cmplx) instead.

- **VlocalCutoff**
Section: Hamiltonian
Type: integer
Default: value of PeriodicDimensions

Define which cutoff type is to be applied to the long range part of the local potential A cutoff is used when one wants to avoid the long range interactions among the system enclosed in the simulation box and (some of) its periodic images

Options:

- **cutoff_sphere**: Cut off the interaction out of a sphere
- **cutoff_cylinder** (1): Cut off the interaction out of a cylinder with axis parallel to the x direction
- **cutoff_slab** (2): Cut off the interaction out of a slab in the xy plane
- **cutoff_none** (3): Do not apply any cutoff: all the periodic images interact

- **VlocalCutoffRadius**
Section: Hamiltonian
Type: float
Default: value of the largest nonperiodic box length

The maximum length out of which the long range part of the interaction is cut off. It refers to the radius of the cylinder if VlocalCutoff = 1, to the thickness od the slab if VlocalCutoff = 2.

6.3.1 Poisson

- **PoissonSolver**
Section: Hamiltonian::Poisson
Type: integer
Default: fft

Defines which method to use in order to solve the Poisson equation. The default for 1D and 2D is the direct evaluation of the Hartree potential.

Options:

- **direct1D** (-1): Direct evaluation of the Hartree potential (in 1D)
- **direct2D** (-2): Direct evaluation of the Hartree potential (in 2D)
- **fft**: FFTs using spherical cutoff (in 2D or 3D; uses FFTW)
- **fft_cyl** (1): FFTs using cylindrical cutoff (in 3D; uses FFTW)
- **fft_pla** (2): FFTs using planar cutoff (in 3D; uses FFTW)
- **fft_nocut** (3): FFTs without using a cutoff (in 3D; uses FFTW)
- **fft_corrected** (4): FFTs + corrections
- **cg** (5): Conjugated gradients
- **cg_corrected** (6): Corrected conjugated gradients
- **multigrid** (7): Multigrid method

- **PoissonSolverMaxMultipole**
Section: Hamiltonian::Poisson
Type: integer

Order of the multipolar expansion for boundary corrections. Default is 4 for cg_corrected and multigrid and 2 for fft_corrected.

- **PoissonSolverThreshold**
  Section: Hamiltonian::Poisson
  Type: integer

  The tolerance for the poisson solution, used by the cg and multigrid solvers. Default is 10^{-5}.

6.3.1.1 Multigrid

- **PoissonSolverMGMaxCycles**
  Section: Hamiltonian::Poisson::Multigrid
  Type: integer
  Default: 20

  Maximum number of multigrid cycles that are performed if convergence is not achieved
- **PoissonSolverMGPostsmoothingSteps**
  Section: Hamiltonian::Poisson::Multigrid
  Type: integer
  Default: 3

  Number of gauss-seidel smoothing steps after coarse level correction in multigrid Poisson solver
- **PoissonSolverMGPresmoothingSteps**
  Section: Hamiltonian::Poisson::Multigrid
  Type: integer
  Default: 3

  Number of gauss-seidel smoothing steps before coarse level correction in multigrid Poisson solver.
- **PoissonSolverMGRelaxationFactor**
  Section: Hamiltonian::Poisson::Multigrid
  Type: float
  Default: 1.0

  Relaxation factor of the relaxation operator used for the multigrid method, default is 1.0. This is mainly for debugging, overrelaxation does nos help in a multigrid scheme.
- **PoissonSolverMGRelaxationMethod**
  Section: Hamiltonian::Poisson::Multigrid
  Type: integer
  Default: Gauss-Seidel

  Method used to solve the linear system approximately in each grid for the multigrid procedure that solve Poisson equation. For the moment, the option conjugate gradients is experimental.

  Options:

  – **gauss_seidel** (1): Gauss-Seidel

  - **gauss_jacobi** (2): Gauss-Jacobi
  - **cg** (5): Conjugate-gradients
- **PoissonSolverMGRestrictionMethod**
  Section: Hamiltonian::Poisson::Multigrid
  Type: integer
  Default: fullweight

  Method used from fine to coarse grid transfer.
  Options:
  - **injection** (1): Injection
  - **fullweight** (2): Fullweight restriction

6.3.2 XC

- **CFunctional**
  Section: Hamiltonian::XC
  Type: integer
  Default: lda_c_pz

  Defines the correlation functional
  Options:
  - **gga_c_pbe** (102): Perdew, Burke & Ernzerhof correlation
  - **lda_c_pw** (10): LDA: Perdew & Wang
  - **lda_c_ob_pw** (11): LDA: Ortiz & Ballone (PW-type parametrization)
  - **lda_c_lyp** (12): LDA: Lee, Yang, & Parr LDA
  - **lda_c_amgb** (13): LDA Attacalite et al functional for the 2D electron gas
  - **mgga_c_tpss** (202): MGGA (not working)
  - **lda_c_wigner** (2): LDA: Wigner parametrization
  - **lda_c_rpa** (3): LDA: Random Phase Approximation
  - **lda_c_hl** (4): LDA: Hedin & Lundqvist
  - **lda_c_gl** (5): LDA: Gunnarson & Lundqvist
  - **lda_c_xalpha** (6): LDA: Slater s Xalpha
  - **lda_c_vwn** (7): LDA: Vosko, Wilk, & Nussair
  - **lda_c_pz** (8): LDA: Perdew & Zunger
  - **lda_c_ob_pz** (9): LDA: Ortiz & Ballone (PZ-type parametrization)
- **JFunctional**
  Section: Hamiltonian::XC
  Type: integer
  Default: lca_omc

  Defines the current functional
  Options:
  - **lca_omc** (301): Orestes, Marcasso & Capelle

  - **lca_lch** (302): Lee, Colwell & Handy
- **OEP_level**
  Section: Hamiltonian::XC
  Type: integer
  Default: oep_none

  At what level shall octopus handle the OEP equation
  Options:
  - **oep_none** (1): Do not solve OEP equation
  - **oep_slater** (2): Slater approximation
  - **oep_kli** (3): KLI approximation
  - **oep_ceda** (4): CEDA (common energy denominator) approximation (not implemented)
  - **oep_full** (5): Full solution of OEP equation using the approach of S. Kuemmel (half implemented)
- **SICorrection**
  Section: Hamiltonian::XC
  Type: integer
  Default: sic_none

  This variable controls which Self Interaction Correction to use. Note that this correction will be applyed to the functional chosen by 'XFunctional' and 'CFunctional'
  Options:
  - **sic_none** (1): No Self Interaction Correction
  - **sic_pz** (2): SIC a Perdew Zunger, hadled by the OEP technique
  - **sic_amaldi** (3): Amaldi correction term (NOT WORKING)
- **XFunctional**
  Section: Hamiltonian::XC
  Type: integer
  Default: lda_x

  Defines the exchange functional
  Options:
  - **gga_x_pbe** (101): GGA: Perdew, Burke & Ernzerhof (GGA)
  - **gga_xc_lb** (103): GGA: van Leeuwen & Baerends (GGA)
  - **lda_x** (1): LDA
  - **mgga_x_tpss** (201): MGGA (not working)
  - **oep_x** (401): OEP: Exact exchange

6.3.3 Casida

- **LinearResponseKohnShamStates**
  Section: Linear Response::Casida
  Type: string

The calculation of the excitation spectrum of a system in the frequency-domain formulation of linear-response time-dependent density functional theory (TDDFT) implies the use of a basis set of occupied/unoccupied Kohn-Sham orbitals. This basis set should, in principle, include all pairs formed by all occupied states, and an infinite number of unoccupied states. In practice, one has to truncate this basis set, selecting a number of occupied and unoccupied states that will form the pairs. These states are specified with this variable. If there are, say, 10 occupied states, and one sets this variable to the value "10-18", this means that occupied states from 10 to 15, and unoccupied states from 16 to 18 will be considered.

This variable is a string in list form, i.e. expressions such as "1,2-5,8-15" are valid. You should include a non-null number of unoccupied states and a non-null number of occupied states.

6.3.4 Static Polarization

- **POLStaticField**
  Section: Linear Response::Static Polarization
  Type: float
  Default: 0.01 a.u.

  Magnitude of the static field used to calculate the static polarizability (`CalculationMode = pol`)

6.3.5 Vibrational Modes

- **Displacement**
  Section: Linear Response::Vibrational Modes
  Type: float
  Default: 0.01 a.u.

  When calculating phonon properties by finite differences (`CalculationMode = phonons`) `Displacement` controls how much the atoms are to be moved in order to calculate the dynamical matrix.

6.3.6 General

- **ODESolver**
  Section: Math::General
  Type: integer
  Default: ode_rk4

  Specifies what kind of ODE solver will be used.

  Options:
  - **ode_rk4** (1): Standard Runge-Kutta 4th order
  - **ode_fb78** (2): Fehlberg solver
  - **ode_vr89** (3): Verner solver
  - **ode_pd89** (4): Prince-Dormand solver

- **ODESolverNSteps**
  Section: Math::General
  Type: integer
  Default: 100

  Number of steps which the chosen ODE solver should perform in the integration interval [a,b] of the ODE.

- **RootSolver**
  Section: Math::General
  Type: integer
  Default: root_newton

  Specifies what kind of root solver will be used.

  Options:

  - **root_bisection** (1): Bisection method
  - **root_brent** (2): Brent method
  - **root_newton** (3): Newton method
  - **root_laguerre** (4): Laguerre method
  - **root_watterstrom** (5): Watterstrom method

- **RootSolverAbsTolerance**
  Section: Math::General
  Type: float
  Default: 1e-8

  Relative tolerance for the root finding process.

- **RootSolverHavePolynomial**
  Section: Math::General
  Type: logical
  Default: no

  If set to yes, the coefficients of the polynomial have to be passed to the root solver.

- **RootSolverMaxIter**
  Section: Math::General
  Type: integer
  Default: 100

  In case of an interative root solver this variable determines the maximal number of iteration steps.

- **RootSolverRelTolerance**
  Section: Math::General
  Type: float
  Default: 1e-8

  Relative tolerance for the root finding process.

- **RootSolverWSRadius**
  Section: Math::General
  Type: float
  Default: 1.0

  Radius of circle in the complex plane. If RootSolverWSRadius = 1.0 the unit roots of a n-th order polynomial are taken as initial values.
- **ScalarMeshType**
  Section: Math::General
  Type: integer
  Default: mesh_sinh

  Specifies what kind of scalar mesh will be used
  Options:
  - **mesh_linear** (1): Linear mesh
  - **mesh_double_log** (2): Logarithmic mesh
  - **mesh_log** (3): Double logarithmic mesh
  - **mesh_sinh** (4): Sinh mesh
  - **gauss_legendre** (5): Gauss-Legendre mesh
- **SparskitAbsTolerance**
  Section: Math::General
  Type: float
  Default: 1e-8

  Some Sparskit solver use an absolute tolerance as stopping criteria for the iteratve solution process. This variable can be used to specify the tolerance.
- **SparskitKrylovSubspaceSize**
  Section: Math::General
  Type: integer
  Default: 15

  Some of the Sparskit solver are Krylov subspace methods. This variable determines which size the solver will use for the subspace.
- **SparskitMaxIter**
  Section: Math::General
  Type: integer

  This variable controls the maximum number of iteration steps that will be performed by the (iterative) linear solver.
- **SparskitPreconditioning**
  Section: Math::General
  Type: integer

  This variable determines what kind of preconditioning the chosen Sparskit solver will use. However, currently there is none implemented.

- **SparskitRelTolerance**
  Section: Math::General
  Type: float
  Default: 1e-8

  Some Sparskit solver use a relative tolerance as stopping criteria for the iteratve solution process. This variable can be used to specify the tolerance.
- **SparskitSolver**
  Section: Math::General
  Type: integer
  Default: sk_cg

  Specifies what kind of linear solver will be used
  Options:
  - **sk_dqgmres** (10): Direct versions of Quasi Generalize Minimum Residual method
  - **sk_cg** (1): Conjugate Gradient Method
  - **sk_cgnr** (2): Conjugate Gradient Method (Normal Residual equation)
  - **sk_bcg** (3): Bi-Conjugate Gradient Method
  - **sk_dbcg** (4): BCG with partial pivoting
  - **sk_bcgstab** (5): BCG stabilized
  - **sk_tfqmr** (6): Transpose-Free Quasi-Minimum Residual method
  - **sk_fom** (7): Full Orthogonalization Method
  - **sk_gmres** (8): Generalized Minimum Residual method
  - **sk_fgmres** (9): Flexible version of Generalized Minimum Residual method
- **WatterstromODESolver**
  Section: Math::General
  Type: integer
  Default: ode_pd89

  The Watterstrom method (cf. J. Comp. Phys., 8, (1971), p. 304-308) transforms the root finding for n-th order polynomials into the solution of n uncoupled ODEs. This variable specifies the ODESolver that should be used for the ODE stepping. Valid solver types are the ones that are allowed for ODESolver (cf. variable ODESolver).
  Options:
  - **ode_rk4** (1): Standard Runge-Kutta 4th order
  - **ode_fb78** (2): Fehlberg solver
  - **ode_vr89** (3): Verner solver
  - **ode_pd89** (4): Prince-Dormand solver
- **WatterstromODESolverNSteps**
  Section: Math::General
  Type: integer
  Default: 400

Number of steps which the chosen ODE solver should perform in the integration interval [a,b] of the Watterstrom ODE.

6.3.7 Curvilinear

- **CurvMethod**
  Section: Mesh::Curvilinear
  Type: integer
  Default: curv_uniform

  The relevant functions in octopus are represented on a mesh in real space. This mesh may be an evenly spaced regular rectangular grid (standard mode), or else an *adaptive* or *curvilinear grid*. We have implemented three kinds of adative meshes, although only one is currently working, the one invented by F. Gygi ("curv_gygi"). The code will stop if any of the other two is invoked.

  Options:

  - **curv_uniform** (1): Regular, uniform rectangular grid. The default.
  - **curv_gygi** (2): The deformation of the grid is done according to the scheme described by F. Gygi [F. Gygi and G. Galli, Phys. Rev. B 52, R2229 (1995)]
  - **curv_briggs** (3): The deformation of the grid is done according to the scheme described by Briggs [E.L. Briggs, D.J. Sullivan, and J. Bernholc, Phys. Rev. B 54 14362 (1996)] (NOT WORKING)
  - **curv_modine** (4): The deformation of the grid is done according to the scheme described by Modine [N.A. Modine, G. Zumbach and E. Kaxiras, Phys. Rev. B 55, 10289 (1997)] (NOT WORKING)

6.3.7.1 Gygi

- **CurvGygiA**
  Section: Mesh::Curvilinear::Gygi
  Type: float

  The grid spacing is reduced locally around each atom, and the reduction is given by 1/(1+A), where A is specified by this variable, CurvGygiA. So, if A=1/2 (the default), the grid spacing is reduced to two thirds = 1/(1+1/2). [This is the A_α variable in Eq. 2 of F. Gygi and G. Galli, Phys. Rev. B 52, R2229 (1995)] It must be larger than zero.

- **CurvGygiAlpha**
  Section: Mesh::Curvilinear::Gygi
  Type: float

  This number determines the region over which the grid is enhanced (range of enhancement of the resolution). That is, the grid is enhanced on a sphere around each atom, whose radius is given by this variable. [This is the a_α variable in Eq. 2 of F. Gygi and G. Galli, Phys. Rev. B 52, R2229 (1995)]. The default is two atomic units. It must be larger than zero.

- **CurvGygiBeta**
  Section: Mesh::Curvilinear::Gygi

Type: float

This number determines the distance over which Euclidean coordinates are recovered. [This is the b_α variable in Eq. 2 of F. Gygi and G. Galli, Phys. Rev. B 52, R2229 (1995)]. The default is four atomic units. It must be larger than zero.

6.3.8 Derivatives

- **DerivativesSpace**
  Section: Mesh::Derivatives
  Type: integer
  Default: real_space

  Defines in which space the gradients and the Laplacian are calculated.

  Options:
  - **real_space**: Derivatives are calculated in real-space using finite differences.
  - **fourier_space** (1): Derivatives are calculated in reciprocal space. Obviously this case implies cyclic boundary conditions, so be careful.

- **DerivativesStencil**
  Section: Mesh::Derivatives
  Type: integer
  Default: stencil_star

  Decides what kind of stencil is used, i.e. what points, around each point in the mesh, are the neighboring points used in the expression of the differential operator.

  If curvilinear coordinates are to be used, then only the "stencil_starplus" or the "stencil_cube" may be used. We only recommend the "stencil_starplus", the cube typically needing way too much memory resources.

  Options:
  - **stencil_star** (1): A star around each point (i.e., only points in the axis).
  - **stencil_variational** (2): Same as the star, but with coefficients built in a different way.
  - **stencil_cube** (3): A cube of points around each point.
  - **stencil_starplus** (4): The star, plus a number of off-axis points.

6.3.9 FFTs

- **DoubleFFTParameter**
  Section: Mesh::FFTs
  Type: float
  Default: 2.0

  For solving Poisson equation in Fourier space, and for applying the local potential in Fourier space, an auxiliary cubic mesh is built. This mesh will be larger than the circumscribed cube to the usual mesh by a factor `DoubleFFTParameter`. See the section that refers to Poisson equation, and to the local potential for details [The default value of two is typically good].

- **FFTOptimize**
  Section: Mesh::FFTs
  Type: logical
  Default: yes

  Should `octopus` optimize the FFT dimensions? This means that the cubic mesh to which FFTs are applied is not taken to be as small as possible: some points may be added to each direction in order to get a "good number" for the performance of the FFT algorithm. In some cases, namely when using the split-operator, or Suzuki-Trotter propagators, this option should be turned off.

6.3.10 Simulation Box

- **BoxShape**
  Section: Mesh::Simulation Box
  Type: integer
  Default: minimum

  This variable decides the shape of the simulation box. Note that some incompatibilities apply:
  - Spherical or minimum mesh is not allowed for periodic systems.
  - Cylindrical mesh is not allowed for systems that are periodic in more than one dimension.

  Options:
  - **sphere** (1): The simulation box will be a sphere of radius Radius
  - **cylinder** (2): The simulation box will be a cylinder with radius Radius and height two times Xlength
  - **minimum** (3): The simulation box will be constructed by adding spheres created around each atom (or user defined potential), of radius Radius.
  - **parallelepiped** (4): The simulation box will be a parallelpiped whose dimensions are taken from the variable lsize.
- **Lsize**
  Section: Mesh::Simulation Box
  Type: block

  In case BoxShape is "parallelepiped", this is assumed to be a block of the form:

  ```
  %Lsize
    sizex | sizey | sizez
  %
  ```

  where the "size*" are half the lengths of the box in each direction.

  If BoxShape is "parallelpiped", this block has to be defined in the input file. The number of columns must match the dimensionality of the calculation.
- **PeriodicDimensions**
  Section: Mesh::Simulation Box
  Type: integer

Define which directions are to be considered periodic. Of course, it has to be a number from 0 to three, and it cannot be larger than Dimensions.

Options:

- **0**: No direction is periodic (molecule)
- **1**: The x direction is periodic (wire)
- **2**: The x and y directions are periodic (slab)
- **3**: The x, y, and z directions are periodic (bulk)

- **Radius**
Section: Mesh::Simulation Box
Type: float

If BoxShape is not "parallelepiped" defines the radius of the spheres or of the cylinder. It has to be a positive number. If it is not defined in the input file, then the program will attempt to fine a suitable default, but this is not always possible, in which case the code will stop issuing this error message.

- **Xlength**
Section: Mesh::Simulation Box
Type: float

If BoxShape is "cylinder", it is half the total length of the cylinder.

6.4 Output

- **Output**
Section: Output
Type: flag
Default: no

Specifies what to print. The output files go into the "static" directory, except when running a time-dependent simulation, when the directory "td.XXXXXXX" is used. Example: "density + potential"

Options:

- **geometry** (16): Outputs a XYZ file called "geometry.xyz" containing the coordinates of the atoms treated within Quantum Mechanics. If point charges were defined in the PDB file (see PDBCoordinates), they will be output in the file "geometry_classical.xyz".
- **potential** (1): Prints out Kohn-Sham potential, separated by parts. File names would be "v0" for the local part, "vc" for the classical potential (if it exists), "vh" for the Hartree potential, and "vxc-x" for each of the exchange and correlation potentials of a give spin channel, where "x" stands for the spin channel.
- **density** (2): Prints out the density. The output file is called "density-i", where "i" stands for the spin channel.
- **ELF** (32): Prints out the electron localization function, ELF. The output file is called "elf-i", where i stands for the spin channel.

  - **wfs** (4): Prints out wave-functions. Which wavefunctions are to be printed is specified by the variable `OutputWfsNumber` – see below. The output file is called "wf-k-p-i", where k stands for the k number, p for the state, and i for the spin channel.
  - **ELF_FS** (64): Prints the electron localization function in Fourier space. The output file is called "elf_FS-i", where i stands for the spin channel. (EXPERIMENTAL)
  - **wfs_sqmod** (8): Prints out squared module of wave-functions. The output file is called "sqm-wf-k-p-i", where k stands for the k number, p for the state, and i for the spin channel.

- **OutputBandsMode**
  Section: Output
  Type: integer
  Default: 1024

  Chose if the band file is to be written in gnuplot or xmgrace friendly format
  Options:
  - **gnuplot** (1024): gnuplot format
  - **xmgrace** (2048): xmgrace format

- **OutputHow**
  Section: Output
  Type: flag

  Describes the format of the output files (see `Output`). Example: "axis_x + plane_x + dx"
  Options:
  - **gnuplot** (1024): Adds newlines to the plane cuts, so that gnuplot can print them in 3D
  - **dx_cdf** (128): Outputs in NetCDF (http://www.unidata.ucar.edu/packages/netcdf/) format. This file can then be read, for example, by OpenDX. The string ".ncdf" is appended to previous file names. Requires the NetCDF library.
  - **plane_y** (16): A plane slice at $y = 0$ is printed. The string ".y=0" is appended to previous file names.
  - **axis_x** (1): The values of the function on the x axis are printed. The string ".y=0,z=0" is appended to previous file names.
  - **plain** (256): Restart files are output in plain binary.
  - **axis_y** (2): The values of the function on the y axis are printed. The string ".x=0,z=0" is appended to previous file names.
  - **plane_z** (32): A plane slice at $y = 0$ is printed. The string ".z=0" is appended to previous file names.
  - **axis_z** (4): The values of the function on the z axis are printed. The string ".x=0,y=0" is appended to previous file names.
  - **mesh_index** (512): Generates output files of a given quantity (Density, Wfs, ...) which include the internal numbering of mesh points. Since this

mode produces large datafiles this is only useful for small meshes and debugging purposes. The output can also be used to display the mesh directly. A gnuplot script for mesh vizualization can be found under `PREFIX/share/octopus/util/display_mesh_index.gp`

- **dx** (64): For printing all the three dimensional information, the open source program visualization tool OpenDX (http://www.opendx.org/) is used. The string ".dx" is appended to previous file names.
- **plane_x** (8): A plane slice at $x = 0$ is printed. The string ".x=0" is appended to previous file names.

- **OutputWfsNumber**
Section: Output
Type: string
Default: "1-1024"

Which wavefunctions to print, in list form, i.e., "1-5" to print the first five states, "2,3" to print the second and the third state, etc.

6.5 SCF

- **GuessMagnetDensity**
Section: SCF
Type: integer
Default: ferromagnetic

The guess density for the SCF cycle is just the sum of all the atomic densities. When performing spin-polarized or non-collinear spin calculations this option sets the guess magnetization density.

For anti-ferromagnetic configurations the `user_defined` option should be used.

Note that if the `paramagnetic` option is used the final ground-state will also be paramagnetic, but the same is not true for the other options.

Options:

- **paramagnetic** (1): Magnetization density is zero.
- **ferromagnetic** (2): Magnetization density is the sum of the atomic magnetization densities.
- **random** (3): Each atomic magnetization density is randomly rotated.
- **user_defined** (4): The atomic magnetization densities are rotated so that the magnetization vector has the same direction as a vector provided by the user. In this case, the `AtomsMagnetDirection` block has to be set.

- **LCAOStart**
Section: SCF
Type: logical
Default: yes

Before starting a SCF calculation, performs a LCAO calculation. These should provide

octopus with a good set of initial wave-functions, and help the convergence of the SCF cycle. (Up to current version, only a minimal basis set used.)

- **SCFinLCAO**
  Section: SCF
  Type: logical
  Default: no

  Performs all the SCF cycle restricting the calculation to the LCAO subspace. This may be useful for systems with convergence problems (first do a calculation within the LCAO subspace, then restart from that point for an unrestricted calculation).

6.5.1 Convergence

- **ConvAbsDens**
  Section: SCF::Convergence
  Type: float
  Default: 1e-5

  Absolute convergence of the density: $\epsilon = \int \mathrm{d}^3 r (\rho^{out}(\mathbf{r}) - \rho^{\mathbf{inp}}(\mathbf{r}))^{\mathbf{2}}$. A zero value means do not use this criterion.

- **ConvAbsEv**
  Section: SCF::Convergence
  Type: float
  Default: 0.0

  Absolute convergence of the eigenvalues: $\epsilon = \sum_{j=1}^{N_{occ}} |\epsilon_j^{out} - \epsilon_j^{inp}|$. A zero value means do not use this criterion.

- **ConvRelDens**
  Section: SCF::Convergence
  Type: float
  Default: 0.0

  Relative convergence of the density: $\epsilon = \frac{1}{N} \int \mathrm{d}^3 r (\rho^{out}(\mathbf{r}) - \rho^{\mathbf{inp}}(\mathbf{r}))^{\mathbf{2}}$. N is the total number of electrons in the problem. A zero value means do not use this criterion.

- **ConvRelEv**
  Section: SCF::Convergence
  Type: float
  Default: 0.0

  Relative convergence of the eigenvalues: $\epsilon = \frac{1}{N} \sum_{j=1}^{N_{occ}} |\epsilon_j^{out} - \epsilon_j^{inp}|$. N is the total number of electrons. A zero value means do not use this criterion._m

- **MaximumIter**
  Section: SCF::Convergence
  Type: integer
  Default: 200

Maximum number of SCF iterations. The code will stop even if convergence has not been achieved. 0 means unlimited.

6.5.2 EigenSolver

- **EigenSolver**
  Section: SCF::EigenSolver
  Type: integer
  Default: cg

  Decides the eigensolver that obtains the lowest eigenvalues and eigenfunctions of the Kohn-Sham Hamiltonian.

  Options:
  - **trlan** (1): Lanczos scheme. Requiers the TRLan package.
  - **plan** (2): Preconditioned Lanczos scheme.
  - **arpack** (3): Implicitly Restarted Arnoldi Method. Requires the ARPACK package
  - **cg** (5): Conjugate-gradients algorithm
  - **jdqz** (5): Jacobi-Davidson scheme. Requires the JDQZ package
  - **cg_new** (6): A rewritting of the cg option, that will eventually substitute it.
  - **evolution** (7): Propagation in imaginary time
- **EigenSolverArnoldiVectors**
  Section: SCF::EigenSolver
  Type: integer
  Default: 20

  This indicates how many Arnoldi vectors are generated It must satisfy EigenSolver-ArnoldiVectors - Number Of Eigenvectors >= 2. See the ARPACK documentation for more details.
- **EigenSolverFinalTolerance**
  Section: SCF::EigenSolver
  Type: float
  Default: 1.0e-6

  This is the final tolerance for the eigenvectors. Must be smaller than EigenSolverInitTolerance.
- **EigenSolverFinalToleranceIteration**
  Section: SCF::EigenSolver
  Type: integer
  Default: 7

  Determines how many interactions are needed to go from EigenSolverInitTolerance to EigenSolverFinalTolerance. Must be larger than 1.
- **EigenSolverImaginaryTime**
  Section: SCF::EigenSolver
  Type: float

Default: 1.0

The imaginary-time step that is used in the imaginary-time evolution method to obtain the lowest eigenvalues/eigenvectors. It must satisfy EigenSolverImaginaryTime > 0.

- **EigenSolverInitTolerance**
  Section: SCF::EigenSolver
  Type: float
  Default: 1.0e-6

  This is the initial tolerance for the eigenvectors.

- **EigenSolverMaxIter**
  Section: SCF::EigenSolver
  Type: integer
  Default: 25

  Determines the maximum number of iterations for the eigensolver (per state) – that is, if this number is reached, the diagonalization is stopped even if the desired tolerance was not achieved. Must be larger or equal than 1.

6.5.3 Mixing

- **MixNumberSteps**
  Section: SCF::Mixing
  Type: integer
  Default: 3

  In the Broyden and in the GR-Pulay scheme, the new input density or potential is constructed from the values of densities/potentials of previous a given number of previous iterations. This number is set by this variable.

- **Mixing**
  Section: SCF::Mixing
  Type: float
  Default: 0.3

  Both the linear and the Broyden scheme depend on a "mixing parameter", set by this variable.

- **TypeOfMixing**
  Section: SCF::Mixing
  Type: integer

  The scheme scheme used to produce, at each iteration in the self consistent cycle that attempts to solve the Kohn-Sham equations, the input density from the value of the input and output densities of previous iterations.

  Options:

  – **linear**: Simple linear mixing.

  – **gr_pulay** (1): "Guaranteed-reduction" Pulay scheme [D. R. Bowler and M. J.

Gillan, Chem. Phys. Lett. 325, 473 (2000)].
  - **broyden** (2): Broyden scheme [C. G Broyden, Math. Comp. 19, 577 (1965); D. D. Johnson, Phys. Rev. B 38, 12807 (1988)].

- **What2Mix**
  Section: SCF::Mixing
  Type: integer
  Default: density

  Selects what should be mixed during the SCF cycle.

  Options:
  - **potential** (1): The Kohn-Sham potential
  - **density** (2): The density

6.6 States

- **CenterOfInversion**
  Section: States
  Type: integer
  Default: no

  Only used in 1D periodic calculation to enforce the correspondig symmetry in the Brillouin Zone

  Options:
  - **no**: The system has no center of inversion: use the whole BZ
  - **yes** (1): The system has a center of inversion: use half BZ

- **ElectronicTemperature**
  Section: States
  Type: float
  Default: 0.0

  If `Occupations` is not set, `ElectronicTemperature` is the temperature in the Fermi-Dirac function used to distribute the electrons among the existing states.

- **ExcessCharge**
  Section: States
  Type: float
  Default: 0.0

  The net charge of the system. A negative value means that we are adding electrons, while a positive value means we are taking electrons from the system.

- **ExtraStates**
  Section: States
  Type: integer

  The number of states is in principle calculated considering the minimum numbers of states necessary to hold the electrons present in the system. The number of electrons is

in turn calculated considering the nature of the species supplied in the `Species` block, and the value of the `ExcessCharge` variable. However, one may command `octopus` to put more states, which is necessary if one wants to use fractional occupational numbers, either fixed from the origin through the `Occupations` block or by prescribing an electronic temperature with `ElectronicTemperature`.

Note that this number is unrelated to `CalculationMode == unocc`.

- **NumberKPoints**
  Section: States
  Type: block
  Default: 1,1,1

  A triplet of integers defining the number of kpoints to be used along each direction in the reciprocal space. The numbers refer to the whole BZ, and the actual number of kpoints is usually reduced exploiting the symmetries of the system For example, the following input samples the BZ with 100 points in the xy plane of the reciprocal space

```
%NumberKPoints
  10 | 10 | 1
%
```

- **Occupations**
  Section: States
  Type: block

  The occupation numbers of the orbitals can be fixed through the use of this variable. For example:

```
%Occupations
  2.0 | 2.0 | 2.0 | 2.0 | 2.0
%
```

  would fix the occupations of the five states to *2.0*. There must be as many columns as states in the calculation. If `SpinComponents == polarized` this block should contain two lines, one for each spin channel. This variable is very useful when dealing with highly symmetric small systems (like an open shell atom), for it allows us to fix the occupation numbers of degenerate states in order to help `octopus` to converge. This is to be used in conjuction with `ExtraStates`. For example, to calculate the carbon atom, one would do:

```
ExtraStates = 2
%Occupations
  2 | 2/3 | 2/3 | 2/3
%
```

- **ShiftKpoints**
  Section: States
  Type: block
  Default: 0.0,0.0,0.0

  A triplet of real numbers to shift the Monkhorst-Pack k-points grid from its default position

- **SpinComponents**
  Section: States
  Type: integer
  Default: unpolarized

  The calculations may be done in three different ways: spin-restricted (TD)DFT (i.e., doubly occupied "closed shells"), spin-unsrestricted or "spin-polarized" (TD)DFT (i.e. we have two electronic systes, one with spin up and one with spin down), or making use of two-component spinors.

  Options:

  – **unpolarized** (1): Spin-restricted calculations.
  – **spin_polarized** (2): Spin unrestricted, also know as spin-DFT, SDFT. This mode will double the number of wave functions will double the number of wave-functions necessary for a spin-unpolarised calculation.
  – **spinors** (3): The spin-orbitals are two-component spinors. This effectively allows the spin-density to arrange non-collinearly - i.e. the magnetization vector is allowed to take different directions in different points.

- **UserDefinedStates**
  Section: States
  Type: block

  Instead of using the ground state as initial state for time propagations it might be interesting in some cases to specify alternative states. Similar to user defined potentials this block allows to specify formulas for the orbitals at t=0.

  Example:

```
%UserDefinedStates
  1 | 1 | 1 | "exp(-r^2)*exp(-i*0.2*x)"
%
```

  The first column specifies the component of the spinor, the second column the number of the state and the third contains kpoint and spin quantum numbers. Finally column four contains a formula for the correspondig orbital.

  Octopus reads first the ground state orbitals from the restart_gs directory. Only the states that are specified in the above block will be overwritten with the given analytical expression for the orbital.

6.7 System

- **SystemName**
  Section: System
  Type: string
  Default: "system"

  A string that identifies the current run. This parameter is seldomly used, but it is sometimes useful to have in the input file.

6.7.1 Coordinates

- **Coordinates**
  Section: System::Coordinates
  Type: block

  If neither a "XYZCoordinates" nor a "PDBCoordinates" was found, octopus tries to read the coordinates for the atoms from the block "Coordinates". The format is quite straightforward:

  ```
  %Coordinates
    'C' | -0.56415 | 0.0 | 0.0 | no
    'O' |  0.56415 | 0.0 | 0.0 | no
  %
  ```

  The first line defines a Carbon atom at coordinates ("-0.56415", "0.0", "0.0"), that is _not_ allowed to move during dynamical simulations. The second line has a similar meaning. This block obviously defines a Carbon monoxide molecule, if the input units are AA. Note that in this way it is possible to fix some of the atoms (this is not possible when specifying the coordinates through a "PDBCoordinates" or "XYZCoordinates" file). It is always possible to fix _all_ atoms using the "MoveIons" directive.

- **PDBCoordinates**
  Section: System::Coordinates
  Type: string

  If this variable is present, the program tries to read the atomic coordinates from the file specified by its value. The PDB (Protein Data Bank (http://www.rcsb.org/pdb/)) format is quite complicated, and it goes well beyond the scope of this manual. You can find a comprehensive description in here. From the plethora of instructions defined in the PDB standard, octopus only reads two, "ATOM" and "HETATOM". From these fields, it reads:

  - columns 13-16: The specie; in fact "octopus" only cares about the first letter - "CA" and "CB" will both refer to Carbon - so elements whose chemical symbol has more than one letter can not be represented in this way. So, if you want to run mercury ("Hg") please use one of the other two methods_m to input the coordinates, "XYZCoordinates" or "Coordinates".
  - columns 18-21: The residue. If residue is "QM", the atom is treated in Quantum Mechanics, otherwise it is simply treated as an external classical point charge. Its charge will be given by columns 61-65.
  - columns 31-54: The Cartesian coordinates. The Fortran format is "(3f8.3)".
  - columns 61-65: Classical charge of the atom. The Fortran format is "(f6.2)".

- **XYZCoordinates**
  Section: System::Coordinates
  Type: string

  If "PDBCoordinates" is not present, the program reads the atomic coordinates from the XYZ file specified by the variable "XYZCoordinates" – in case this variable is present.

The XYZ format is very simple: The first line of the file has an integer indicating the number of atoms. The second can contain comments that are simply ignored by "octopus". Then there follows one line per each atom, containing the chemical species and the Cartesian coordinates of the atom.

6.7.2 Species

- **Species**
  Section: System::Species
  Type: block

  A specie is by definition either an "ion" (nucleus + core electrons) described through a pseudo-potential, or an user-defined, model potential.

  Note that some common pseudopotentials are distributed with the code in the directory *OCTOPUS-HOME/share/PP/*. To use these pseudopotentials you are not required to define them explicitly in the Species block, as defaults are provided by the program (you can override these defaults in any case). Additional pseudopotentials can be downloaded from the <a href='http://www.tddft.org/programs/octopus/pseudo.php'>octopus homepage<a>.

  The format of this block is the following: The first field is the name of the specie, followed by the atomic mass (in atomic mass units). The third field defines the type of specie (the valid options are detailed below), each type needs some extra parameters given in the following fields of the row.

  In 3D, e.g.

```
%Species
  'O' | 15.9994 | spec_ps_tm2 | 8 | 1 | 1
  'H' | 1.0079 | spec_ps_hgh | 1 | 0 | 0
  'jlm' | 23.2 | spec_jelli | 8 | 5.0
  'pnt' | 32.3 | spec_point | 2.0
  'udf' | 0.0 | spec_usdef | 8 | "1/2*r^2"
%
```

  Options:

  - **spec_ps_tm2** (100): Troullier Martins pseudopotential, the pseudopotential will be read from an *.ascii* or *.vps* file, either in the working directory or in the *OCTOPUS-HOME/share/PP/TM2* directory. The following three numbers are the atomic number, the maximum l-component of the pseudo-potential to consider in the calculation, and the l-component to consider as local.
  - **spec_ps_hgh** (101): Hartwigsen-Goedecker-Hutter pseudopotentials, the next field is the atomic number an the last two numbers are irrelevant, since they do are not necessary to define the HGH pseudopotentials.
  - **spec_usdef** (1): Specie with user defined potential. In this case, the fourth field is the valence charge and the fifth field is a string with a mathematical expresion that defines the potential (you can use any of the x, y, z or r variables).
  - **spec_point** (2): Point charge, the fourth field is the value of the charge.
  - **spec_jelli** (3): Jellium sphere, the extra parameters are the charge of the jellium sphere (an equal value of valence charge is assumed) and the radius of the sphere.

6.7.3 Velocities

- **RandomVelocityTemp**
  Section: System::Velocities
  Type: string

  If this variable is present, octopus will assign random velocities to the atoms following a Bolzmann distribution with temperature given by RandomVelocityTemp.

- **Velocities**
  Section: System::Velocities
  Type: block

  If XYZVelocities is not present, octopus will try to fetch the initial atomic velocities from this block. If this block is not present, octopus will reset the initial velocities to zero. The format of this block can be illustrated by this example:

  ```
  %Velocities
    'C' | -1.7 | 0.0 | 0.0
    'O' |  1.7 | 0.0 | 0.0
  %
  ```

  It describes one Carbon and one Oxygen moving at the relative velocity of 3.4, velocity units.

  Note: It is important for the velocities to maintain the ordering in which the species were defined in the coordinates specifications.

- **XYZVelocities**
  Section: System::Velocities
  Type: string

  octopus will try to read the starting velocities of the atoms from the XYZ file specified by the variable XYZVelocities. Note that you do not need to specify initial velocities if you are not going to perform ion dynamics; if you are going to allow the ions to move but the velocities are not specified, they are considered to be null.

6.8 Time Dependent

- **TDGauge**
  Section: Time Dependent
  Type: integer
  Default: length

  In which gauge to treat the laser field.

  Options:

  - **length** (1): Length gauge.
  - **velocity** (2): Velocity gauge.

- **TDLasers**
  Section: Time Dependent
  Type: block

The block TDLasers describe the type and shape of time-dependent external perturbations that are applied to the system. Each line of the block describes a laser field; this way you can actually have more than one laser (e.g. a "pump" and a "probe"). The syntax of each line is, then:

```
%TDLasers
  nx | ny | nz | amplitude | omega | envelope | tau0 | t0 | tau1 |
filename1 | filename2
%
```

The first three (possibly complex) numbers mark the polarization direction of the field. The "amplitude" is obviously the amplitude of the field. The "omega" is the frequency. The "envelope" decides the shape of the enveloping function – see the manual for details. "tau0", "t0" and "tau1" are three paramenters that decide on the temporal shape of the pulse – the exact details depend on the particular envelope. If the envelope is given in a file, this will be "filename1". If the spatial part of the field is given in a file, this will be "filename2".

The last three columns ("tau1", "filename1" and "filename2") are optional; they will only be searched if needed.

In order to give the spatial shape of the field in a file, the component "filename2" has to be present. If it is not present, then the laser field will be a dipolar field (which is the usual case).

6.8.1 Absorbing Boundaries

- **ABHeight**
  Section: Time Dependent::Absorbing Boundaries
  Type: float
  Default: -0.2 a.u.

  When `AbsorbingBoundaries == sin2`, is the height of the imaginary potential.
- **ABWidth**
  Section: Time Dependent::Absorbing Boundaries
  Type: float
  Default: 0.4 a.u.

  Width of the region used to apply the absorbing boundaries.
- **AbsorbingBoundaries**
  Section: Time Dependent::Absorbing Boundaries
  Type: integer
  Default: no_absorbing

  To improve the quality of the spectra by avoiding the formation of standing density waves, one can make the boundaries of the simulation box absorbing.

  Options:
  - **no_absorbing**: No absorbing boundaries.
  - **sin2** (1): A $\sin^2$ imaginary potential is added at the boundaries.

  - **mask** (2): A mask is applied to the wave-functions at the boundaries.

6.8.2 Linear Response

- **TDDeltaStrength**
  Section: Time Dependent::Linear Response
  Type: float
  Default: 0.0

  When no laser is applied, a delta (in time) electric field with strength `TDDeltaStrength` is applied. This is used to calculate the linear optical spectra.

- **TDDeltaStrengthMode**
  Section: Time Dependent::Linear Response
  Type: integer
  Default: kick_density

  When calculating the linear response of the density via the propagation in real time, one needs to perfrom an initical kick on the KS system, at time zero. Depending on what kind response property one wants to obtain, this kick may be done in several modes.

  Options:
  - **kick_density**: The total density of the system is perturbed.
  - **kick_spin** (1): The individual spin densities are perturbed differently. Note that this mode is only possible if the run is done in spin polarized mode, or with spinors.
  - **kick_spin_and_density** (2): A combination of the two above. Note that this mode is only possible if the run is done in spin polarized mode, or with spinors.

- **TDPolarization**
  Section: Time Dependent::Linear Response
  Type: block

  The (real) polarization of the delta electric field. Normally one needs three perpendicular polarization directions to calculate a spectrum (unless symmetry is used). The format of the block is:

```
%TDPolarization
  pol1x | pol1y | pol1z
  pol2x | pol2y | pol2z
  pol3x | pol3y | pol3z
%
```

  Octopus uses both this block and the variable `TDPolarizationDirection` to determine the polarization vevtor for the run. For example, if `TDPolarizationDirection=2` the polarization `(pol2x, pol2y, pol2z)` would be used.

  The default value for `TDPolarization` are the three Cartesian unit vectors (1,0,0), (0,1,0), and (0,0,1).

- **TDPolarizationDirection**
  Section: Time Dependent::Linear Response
  Type: integer

Default: 1

Defines which one of the lines of `TDPolarization` to use for the run. In a typical run (without using the symmetry), TDPolarization would contain the three Cartesian unit vectors, and one would make 3 runs varying `TDPolarization` from 1 to 3.

6.8.3 Propagation

- **MoveIons**
  Section: Time Dependent::Propagation
  Type: logical
  Default: static_ions

  What kind of simulation to perform.

  Options:
  - **static_ions**: Do not move the ions.
  - **verlet** (3): Newtonian dynamics using Verlet.
  - **vel_verlet** (4): Newtonian dynamics using velocity Verlet.
- **TDEvolutionMethod**
  Section: Time Dependent::Propagation
  Type: integer
  Default: etrs

  This variable determines which algorithm will be used to approximate the evolution operator $U(t+\delta t, t)$. That is, known $\psi(\tau)$ and $H(\tau)$ for $tau \leq t$, calculate $t+\delta t$. Note that in general the Hamiltonian is not known at times in the interior of the interval $[t, t+\delta t]$. This is due to the self-consistent nature of the time-dependent Kohn-Sham problem: the Hamiltonian at a given time τ is built from the "solution" wavefunctions at that time.

  Some methods, however, do require the knowledge of the Hamiltonian at some point of the interval $[t, t+\delta t]$. This problem is solved by making use of extrapolation: given a number l of time steps previous to time t, this information is used to build the Hamiltonian at arbitrary times within $[t, t+\delta t]$. To be fully precise, one should then proceed *self-consistently*: the obtained Hamiltonian at time $t+\delta t$ may then be used to interpolate the Hamiltonian, and repeat the evolution algorithm with this new information. Whenever iterating the procedure does not change the solution wavefunctions, the cycle is stopped. In practice, in `octopus` we perform a second-order extrapolation without self-consistente check, except for the first two iterations, where obviously the extrapolation is not reliable.

  The proliferation of methods is certainly excessive; The reason for it is that the propagation algorithm is currently a topic of active development. We hope that in the future the optimal schemes are clearly identified. In the mean time, if you do not feel like testing, use the default choices and make sure the time step is small enough.

  Options:
  - **split**: Split Operator (SO). This is one of the most traditional methods. It splits the full Hamiltonian into a kinetic and a potential part, performing the first in

Fourier-space, and the latter in real space. The necessary transformations are performed with the FFT algorithm.

$$U_{\rm SO}(t+\delta t,t) = \exp\{-\frac{i}{2}\delta t T\}\exp\{-i\delta t V^*\}\exp\{-\frac{i}{2}\delta t T\}$$

Since those three exponentials may be calculated exactly, one does not need to use any of the methods specified by variable `TDExponentialMethod` to perform them.

The key point is the specification of V^*. Let $V(t)$ be divided into $V_{\rm int}(t)$, the "internal" potential which depends self-consistently on the density, and $V_{\rm ext}(t)$, the external potential that we know at all times since it is imposed to the system by us (e.g. a laser field): $V(t) = V_{\rm int}(t) + V_{\rm ext}(t)$. Then we define to be V^* to be the sum of $V_{\rm ext}(t+\delta t/2)$ and the internal potential built from the wavefunctions *after* applying the right-most kinetic term of the equation, $\exp\{-i\delta t/2T\}$.

It may the be demonstrated that the order of the error of the algorithm is the same that the one that we would have by making use of the Exponential Midpoint Rule (EM, described below), the SO algorithm to calculate the action of the exponential of the Hamiltonian, and a full self-consistent procedure.

– **suzuki_trotter** (1): This is the generalization of the Suzuki-Trotter algorithm, described as one of the choices of the `TDExponentialMethod`, to time-dependent problem. Consult the paper by O. Sugino and M. Miyamoto, Phys. Rev. B **59**, 2579 (1999), for details.

  It requires of Hamiltonian extrapolations.

– **etrs** (2): The idea is to make use of the time-reversal symmetry from the beginning:

$$\exp\left(-i\delta t/2H_n\right)\psi_n = exp\left(i\delta t/2H_{n+1}\right)\psi_{n+1},$$

  and then invert to obtain:

$$\psi_{n+1} = \exp\left(-i\delta t/2H_{n+1}\right) exp\left(-i\delta t/2H_n\right)\psi_n.$$

  But we need to know H_{n+1}, which can only be known exactly through the solution ψ_{n+1}. What we do is to estimate it by performing a single exponential: $\psi^*_{n+1} = \exp\left(-i\delta t H_n\right)\psi_n$, and then $H_{n+1} = H[\psi^*_{n+1}]$. Thus no extrapolation is performed in this case.

– **aetrs** (3): Approximated Enforced Time-Reversal Symmetry (AETRS). A modification of previous method to make it faster. It is based on extrapolation of the time-dependent potentials. It is faster by about 40%.

  The only difference is the procedure to estimate H_{n+1}: in this case it is extrapolated trough a second-order polynomial by making use of the Hamiltonian at time $t-2\delta t$, $t-\delta t$ and t.

– **exp_mid** (4): Exponential Midpoint Rule (EM). This is maybe the simplest method, but is is very well grounded theretically: it is unitary (if the exponential is performed correctly) and preserves time symmetry (if the self-consistency problem is dealt with correctly). It is defined as:

$$U_{\rm EM}(t+\delta t,t) = \exp\left(-i\delta t H_{t+\delta t/2}\right) .$$

  - **magnus** (5): Magnus Expansion (M4). This is the most sophisticated approach. It is a fourth order scheme (feature that shares with the ST scheme; the other schemes are in principle second order). It is tailored for making use of very large time steps, or equivalently, dealing with problem with very high-frequency time dependence. It is still in a experimental state; we are not yet sure of when it is advantageous.

- **TDExpOrder**
  Section: Time Dependent::Propagation
  Type: integer
  Default: 4

  For `TDExponentialMethod` equal `standard` or `chebyshev`, the order to which the exponential is expanded. For the Lanczos approximation, it is the maximum Lanczos-subspace dimension.

- **TDExponentialMethod**
  Section: Time Dependent::Propagation
  Type: integer
  Default: standard

  Method used to numerically calculate the exponential of the Hamiltonian, a core part of the full algorithm used to approximate the evolution operator, specified through the variable `TDEvolutionMethod`. In the case of using the Magnus method, described below, the action of the exponential of the Magnus operator is also calculated through the algorithm specified by this variable.

  Options:

  - **split**: It is important to distinguish between applying the split operator method to calculate the exponential of the Hamiltonian at a given time – which is what this variable is referring to – from the split operator method as an algorithm to approximate the full evolution operator $U(t+\delta t,t)$, and which will be described below as one of the possibilities of the variable `TDEvolutionMethod`. The equation that describes the split operator scheme is well known:

    $$\exp_{\rm SO}(-i\delta t H) = \exp(-i\delta t/2V)\exp(-i\delta t T)\exp(-i\delta t/2V).$$

    Note that this is a "kinetic referenced SO", since the kinetic term is sandwiched in the middle. This is so because in `octopus`, the states spend most of its time in real-space; doing it "potential referenced" would imply 4 FFTs instead of 2. This split-operator technique may be used in combination with, for example, the exponential midpoint rule as a means to approximate the evolution operator. In that case, the potential operator V that appears in the equation would be calculated at time $t+\delta t/2$, that is, in the middle of the time-step. However, note that if the split-operator method is invoked as a means to approximate the evolution operator (`TDEvolutionMethod = 0`), a different procedure is taken – it will be described below –, and in fact the variable `TDExponentialMethod` has no effect at all.

– **suzuki_trotter** (1): This is a higher-order SO based algorithm. See O. Sugino and Y. Miyamoto, Phys. Rev. B **59**, 2579 (1999). Allows for larger time-steps, but requires five times more time than the normal SO.

The considerations made above for the SO algorithm about the distinction between using the method as a means to approximate $U(t+\delta t)$ or as a means to approximate the exponential also apply here. Setting `TDEvolutionMethod = 1` enforces the use of the ST as an algorithm to approximate the full evolution operator, which is slightly different (see below).

– **lanczos** (2): Allows for larger time-steps. However, the larger the time-step, the longer the computational time per time-step. In certain cases, if the time-step is too large, the code will emit a warning whenever it considers that the evolution may not be properly proceeding – the Lanczos process did not converge. The method consists in a Krylov subspace approximation of the action of the exponential (see M. Hochbruck and C. Lubich, SIAM J. Numer. Anal. **34**, 1911 (1997) for details). Two more variables control the performance of the method: the maximum dimension of this subspace (controlled by variable `TDExpOrder`), and the stopping criterium (controlled by variable `TDLanczosTol`). The smaller the stopping criterium, the more precisely the exponential is calculated, but also the larger the dimension of the Arnoldi subspace. If the maximum dimension allowed by `TDExpOrder` is not enough to meet the criterium, the above-mentioned warning is emitted.

– **standard** (3): This method amounts to a straightforward application of the definition of the exponential of an operator, in terms of it Taylor expansion.

$$\exp_{\mathrm{STD}}(-i\delta t H) = \sum_{i=0}^{k} \frac{(-i\delta t)^i}{i!} H^i.$$

The order k is determined by variable *TDExpOder*. Some numerical considerations (by Jeff Giansiracusa and George F. Bertsch; see http://www.phys.washington.edu/~bertsch/num3.ps) suggest the 4th order as especially suitable and stable.

– **chebyshev** (4): In principle, the Chebyshev expansion of the exponential represents it more accurately than the canonical or standard expansion. As in the latter case, `TDExpOrder` determines the order of the expansion.

There exists a closed analytical form for the coefficients of the exponential in terms of Chebyshev polynomials:

$$\exp_{\mathrm{CHEB}}(-i\delta t H) = \sum_{k=0}^{\infty} (2-\delta_{k0})(-i)^k J_k(\delta t) T_k(H),$$

where J_k are the Bessel functions of the first kind, and H has te be previously scaled to $[-1,1]$. See H. Tal-Ezer and R. Kosloff, J. Chem. Phys. **81**, 3967 (1984); R. Kosloff, Annu. Rev. Phys. Chem. **45**, 145 (1994); C. W. Clenshaw, MTAC **9**, 118 (1955).

- **TDLanczosTol**
  Section: Time Dependent::Propagation

Type: float
Default: 1e-5

An internal tolerance variable for the Lanczos method. The smaller, the more precisely the exponential is calculated, and also the bigger the dimension of the Krylov subspace needed to perform the algorithm. One should carefully make sure that this value is not too big, or else the evolution will be wrong.

- **TDMaximumIter**
Section: Time Dependent::Propagation
Type: integer
Default: 1500

Number of time steps in which the total integration time is divided; in previous notation, N.

- **TDTimeStep**
Section: Time Dependent::Propagation
Type: float
Default: 0.07 a.u.

Time-step for the propagation; in previous notation, δt.

6.8.4 TD Output

- **OutputEvery**
Section: Time Dependent::TD Output
Type: integer
Default: 1000

The output is saved when the iteration number is a multiple of the `OutputEvery` variable.

- **TDDipoleLmax**
Section: Time Dependent::TD Output
Type: integer
Default: 1

Maximum multi-pole of the density output to the file `td.general/multipoles` during a time-dependent simulation. Must be 0 < `TDDipoleLmax` < 5.

- **TDOutput**
Section: Time Dependent::TD Output
Type: flag
Default: multipoles + geometry

Defines what should be output during the time-dependent simulation.

Options:

  - **el_energy** (128): If `set`, `octopus` outputs the different components of the electronic energy to the file `td.general/el_energy`.

- **geometry** (16): If set (and if the atoms are allowed to move), outputs the coordinates, velocities, and forces of the atoms to the the file `td.general/coordinates`.
- **multipoles** (1): Outputs the multipole moments of the density to the file `td.general/multipoles`. This is required to, e.g., calculate optical absorption spectra of finite systems. The maximum value of l can be set with the variable `TDDipoleLmax`.
- **td_occup** (256): If set, outputs the projections of the time-dependent Kohn-Sham wave-functions onto the static (zero time) wave-functions to the file `td.general/projections.XXX`.
- **angular** (2): Outputs the angular momentum of the system that can be used to calculate circular dichroism (EXPERIMENTAL)
- **acceleration** (32): When set outputs the acceleration, calculated from Ehrenfest theorem, in the file `td.general/acceleration`. This file can then be processed by the utility "hs-from-acc" in order to obtain the harmonic spectrum.
- **spin** (4): Outputs the expectation value of the spin, that can be used to calculate magnetic cicular dichroism (EXPERIMENTAL)
- **local_mag_moments** (512): If set, outputs the local magnetic moments, integrated in sphere centered around each atom. The radius of the sphere can be ser with `LocalMagneticMomentsSphereRadius`
- **laser** (64): If set, and if there are lasers defined in `TDLasers`, `octopus` outputs the laser field to the file `td.general/laser`.
- **gs_proj** (8): Outputs the projection of the time-dependent Kohn-Sham Slater determinant onto the ground-state to the file `td.general/gs_projection`. As the calculation of the projection is fairly heavy, this is only done every `OutputEvery` iterations.

6.9 Unoccupied States

- **NumberUnoccStates**
  Section: Unoccupied States
  Type: integer
  Default: 5

  How many unoccupied states to compute.

- **WriteMatrixElements**
  Section: Unoccupied States
  Type: logical
  Default: no

  If true outputs the following matrix elements:

  - $< i|T + V_{ext}|j >$
  - $< ij|1/|r_1 - r_2||kl >$

  in the directory ME

6.9.1 Optical Spectra

- **SpecDampFactor**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

  If `SpecDampMode` is set to "exp", the damping parameter of the exponential is fixed through this variable.

- **SpecDampMode**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

  Decides which damping/filtering is to be applied in order to calculate spectra by calculating a Fourier transform

  Options:

  - **no**: No filtering at all.
  - **exponential** (1): Exponential filtering, corresponding with a Lorentzian-shaped spectrum
  - **polynomial** (2): Third-order polynomial damping.
  - **gaussian** (3): Gaussian damping

- **SpecEndTime**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

  Processing is done for the given function in a time-window that ends at the value of this variable.

- **SpecEnergyStep**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

  Sampling rate for the spectrum.

- **SpecMaxEnergy**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

  The Fourier transform is calculated for energies smaller than this value.

- **SpecStartTime**
  Section: Utilities::Optical Spectra
  Type: integer
  Default: polynomial

Processing is done for the given function in a time-window that starts at the value of this variable.

6.9.2 oct-center-geometry

- **AxisType**
  Section: Utilities::oct-center-geometry
  Type: integer
  Default: inertia

  Besides centering the structure, this is align to a set of orthogonal axis. This variable decides which set of axis to use.

  Options:
  - **inertia** (1): The axis of inertia
  - **pseudo_inertia** (2): Pseudo-axis of inertia, calculated considering that all species have mass one
  - **large_axis** (3): The larger axis of the molecule

6.9.3 oct-harmonic-spectrum

- **HarmonicSpectrumMode**
  Section: Utilities::oct-harmonic-spectrum
  Type: integer
  Default: hs_from_dipole

  The oct-harmonic-spectrum may calculate the spectrum in two alternative ways, mathematically equivalent but numerically diferent: by reading the dipole moment (from the multipoles file) and calculating the accelaratio numerically from it, or by reading directly the acceleration from the acceleration file, which may also be generated during a time-dependent run of octopus.

  Options:
  - **hs_from_dipole** (1): Calculate the harmonic spectrum by numerically differentiating the multipoles file.
  - **hs_from_acceleration** (2): Calculate the harmonic spectrum by reading the acceleration file.
- **HarmonicSpectrumPolarization**
  Section: Utilities::oct-harmonic-spectrum
  Type: string
  Default: `"z"`

  The oct-harmonic-spectrum utility program needs to know the direction along which the emission raidiation is considered to be polarized. It may be linearly polarized or circularly polarized.

  Options:
  - `"+"`: Circularly polarized field, counter clock-wise.

- "-": Circularly polarized field, clock-wise.
- "x": Linearly polarized field in the x direction.
- "y": Linearly polarized field in the y direction.
- "z": Linearly polarized field in the z direction.

7 Undocumented Variables

If you want to use these variables you will have to go to the code to find out what they do. If you do it, please take the time to write a short description and send a patch of the manual to us ;) BTW, some of this variables describe things that may not work, or are under developments, so don't blame us...

- AnimationSampling
- AxisType
- BoundaryZeroDerivative
- CenterOfInversion
- CurrentDFT
- GuessDensityAtomsMagnet
- KineticCutoff
- LB94_beta
- LB94_modified
- LB94_threshold
- OEP_level
- OEP_mixing
- OptControlAlpha
- OptControlEps
- OptControlInitLaser
- OptControlMaxIter
- OutputDuringSCF
- OutputELF_FS
- OutputWfsSqMod
- PeriodicDimensions
- RestartFileFormat
- ShifKPoints
- StaticMagneticField
- TDDeltaStrengthMode
- TDOutputSpin
- VlocalCutoff
- FromScratch

8 External utilities

A few small programs are generated along with octopus, for the purpose of post-processing the generated information. These utilities should all be run from the directory where octopus was run, so that it may see the input file, and the directories created by it.

8.1 oct-sf

This utility generates the dipole strength function of the given system. Its main input is the td.general/multipoles file. Output is written to a file called spectrum. This file is made of two columns: energy (in eV or a.u., depending on the units specified in the input file), and dipole strength function (in 1/eV, or 1/a.u., idem).

In the input file, the user may set the SpecTransformMode – this should be set to "sin" for proper use –, the SpecDampMode – recommended value is "pol", which ensures fulfilling of the N-sum rule, the SpecStartTime, the SpecEndTime, the SpecEnergyStep, the SpecMinEnergy and the SpecMaxEnergy.

8.2 oct-rsf

8.3 oct-hs-mult

Calculates the harmonic spectrum, out of the multipoles file. To do.

8.4 oct-hs-acc

Calculates the harmonic spectrum, out of the acceleration file. To do.

8.5 oct-xyz-anim

Reads out the td.general/coordinates file, and makes a movie in XYZ format. To do.

8.6 oct-excite

Calculates the excitation spectrum within linear response. This utility can output just the difference of eigenvalues by setting LinEigenvalues, the excitations using M. Petersilka formula (LinPetersilka), or M. Casida (LinCasida). This utility requires that a calculation of unoccupied states (CalculationMode = unocc) has been done before, and it outputs the results to the sub-directory "linear". The states entering the calculation can be chosen with the variable ExciteStates.

8.7 oct-broad

Generates a spectrum by broadening the excitations obtained by the oct-excite utility. The parameters of the spectrum can be set using the variables LinBroadening, LinMinEnergy, LinMaxEnergy, and LinEnergyStep.

8.8 oct-make-st

make_st reads tmp/restart.static and replaces some of the Kohn-Sham states by Gaussians wave packets. The states which should be replaced are given in the %MakeStates section in the input file and written to tmp/restart.static.new. (You probably want to copy that file to tmp/restart.static and use then CalculationMode=5 or 6.)

```
%MakeStates
  ik | ist | idim | type | sigma | x0 | k
%
```

The first values stand for

- *ik*: The k point (or the spin, if spin-components=2) of the state
- *ist*: The state to be replaced
- *idim*: The component of the state (if the wave functions have more than one component, i.e. when spin-components=3 is used).
- The *type* of the wave packet; currently only 1 (Gaussian) is available

The next items depend on the type chosen. For a Gaussian wave packet, defined as

$$\psi_0(x) = \frac{1}{\sqrt{2\pi}\sigma} \mathrm{e}^{\mathrm{i}k(x-x_0)} \mathrm{e}^{-\frac{(x-x_0)^2}{2\sigma^2}},$$

they are:

- σ the width of the Gaussian
- k: the **k** vector. In 3D use k1|k2|k3.
- x_0: the coordinate where the Gaussian is initially centred. In 3D use x01|x02|x03.

8.9 oct-center-geom

This utility centers and aligns the coordinates of the molecule. It reads the file specified in the input file (either by the Coordinates block, or by the XYZCoordinates or PDBCoordinates variables), and outputs the adjusted coordinates to the file adjusted.xyz. The following parameters are read from the input file:

- MainAxis (block data): A vector of three reals defining the axis to which the molecule should be align. If not present, the default value will be

  ```
  %MainAxis
    1 | 0 | 1
  %
  ```

- AxisType (int, inertia): This variable describes how to calculate the "main" axis of the molecule. Possible values are:
  - inertia or 1: Axis of inertia;
  - pseudo or 2: Axis of inertia calculated as if all atoms had mass 1;
  - large or 2: The longest axis of the molecule.

8.10 wf.net

This is an OpenDX network, aimed at the visualization of wave-functions. To be able to use it, you need to have properly installed the OpenDX program (get it at opendx.org), as well as the Chemistry extensions obtainable at the Cornell Theory Center (http://www.tc.cornell.edu/Services/Vis/dx/index.asp). Once these are working, you may follow a small tutorial on `wf.net` by following next steps:

- Place in a directory the program wf.net, the (needed) auxiliary file wf.cfg, and the sample inp file that can all be found in `OCTOPUS-HOME`/util.
- Run `octopus`. The inp file used prescribes the calculation of the C atom in its ground state, in spin-polarized mode. It also prescribes that the wave-functions should be written in "dx" format. At the end, these should be written in subdirectory "static": wf-00x-00y-1.dx, where x runs from 1 to 2 (spin quantum number) and y runs from 1 to 4 (wave-function index).
- Run the OpenDX program. Click on "Run Visual Programs" on the DX main menu, and select the program `wf.net`. The program will be executed, and several windows should open. One of them should be called "Application Comment". It contains a small tutorial. Follow it from now on.

9 Examples

9.0.1 Hello world

As a first example, we will take a sodium atom. With your favourite text editor, create the following input "inp".

```
SystemName = 'Na'
CalculationMode = 1
%Species
  'Na' | 22.989768 | 11 | "tm2" | 0 | 0
%
%Coordinates
  'Na' | 0.0 | 0.0 | 0.0 | no
%
Radius = 12.0
Spacing = .6
TypeOfMixing = 2
```

This input file should be essentially self-explanatory. Note that a Troullier-Martins pseudopotential file ("Na.vps", or "Na.ascii") should be accesible to the program. A sample "Na.ascii" may be found in `OCTOPUS-HOME/share/PP/TM2`. If `octopus` was installed (`make install` was issued after `make`), there should be no need to do anything – the program should find it. Otherwise, you may as well place it in the working directory. Then run `octopus` – for example, do `octopus > out` , so that the output is stored in "out" file. If everything goes OK, "out" should look like[1]:

```
                    Running octopus, version 1.1
                (build time - Fri Mar 14 14:23:49 CET 2003)

Info: Calculation started on 2003/03/17 at 03:49:56
Info: Reading pseudopotential from file:
      '/home/marques/share/octopus/PP/TM2/Na.ascii'
      Calculating atomic pseudo-eigenfunctions for specie Na....
      Done.
Info: l =  0 component used as local potential
  Type = sphere           Radius [b] =  12.000
  Spacing [b] = ( 0.600, 0.600, 0.600)    volume/point [b^3] =  0.21600
  # inner mesh =  33401   # outer mesh =  18896
Info: Derivatives calculated in real-space
Info: Local Potential in Reciprocal Space.
Info: FFTs used in a double box (for poisson | local potential)
      box size = (  81,  81,  81)
      alpha =      2.00000
Info: Using FFTs to solve poisson equation with spherical cutoff.
Info: Exchange and correlation
      Exchange    family    : LDA
```

[1] Before this output, a beautiful octopus ascii-art picture may be printed...

```
                functional: non-relativistic
      Correlation family    : LDA
                functional: Perdew-Zunger
Info: Allocating rpsi.
Info: Random generating starting wavefunctions.
Info: Unnormalized total charge =      0.998807
Info: Renormalized total charge =      1.000000
Info: Setting up Hamiltonian.
Info: Performing LCAO calculation.
Info: LCAO basis dimension:      1
      (not considering spin or k-points)
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102098       1.000000
Info: SCF using real wavefunctions.
Info: Broyden mixing used. It can (i) boost your convergence,
      (ii) do nothing special, or (iii) totally screw up the run.
      Good luck!
Info: Converged =     0
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102975       1.000000      (2.8E-02)
Info: iter =    1 abs_dens = 0.53E-03 abs_ener = 0.60E+00

Info: Converged =     0
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102477       1.000000      (1.4E-03)
Info: iter =    2 abs_dens = 0.43E-03 abs_ener = 0.65E-05

Info: Converged =     1
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102419       1.000000      (5.1E-04)
Info: iter =    3 abs_dens = 0.39E-04 abs_ener = 0.20E-06

Info: Converged =     1
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102436       1.000000      (8.5E-05)
Info: iter =    4 abs_dens = 0.24E-04 abs_ener = 0.52E-08

Info: Converged =     1
Eigenvalues [H]
   #   Eigenvalue    Occupation      Error (1)
   1    -0.102437       1.000000      (1.5E-06)
Info: iter =    5 abs_dens = 0.14E-05 abs_ener = 0.36E-10
```

```
Info: SCF converged in    5 iterations
Info: Deallocating rpsi.
Info: Calculation ended on 2003/03/17 at 03:50:04
```

Take now a look at the working directory. It should include the following files:

```
-rw-rw-r--    1 user  group       177 Jul 10 12:29 inp
-rw-rw-r--    1 user  group      4186 Jul 10 12:35 out
-rw-rw-r--    1 user  group      1626 Jul 10 12:35 out.oct
drwxrwxr-x    2 user  group      4096 Jul 10 12:35 static
drwxrwxr-x    2 user  group      4096 Jul 10 12:35 tmp
```

Besides the initial file (`inp`) and the `out` file, two new directories appear. In `static`, you will find the file `info`, with information about the static calculation (it should be hopefully self-explanatory, otherwise please complain to the authors). In `tmp`, you will find the `restart.static`, a binary file containg restart information about the ground-state, which is used if, for example, you want to start a time-dependent calculation afterwards. Finally, you can safely ignore `out.oct`: it is an output from the liboct library, irrelevant for what concerns physics ;).

Exercises:

- Study how the total energy and eigenvalue of the sodium atom improve with the mesh spacing.
- Calculate the static polarizability of the sodium atom (`CalculationMode = 7`). Two new files will be generated: `restart.pol` that can be used to resume the polarizability calculation, and `Na.pol` that contains the static polarizability tensor. Note that this calculation overwrites `tmp/restart.static`, so that what now is there is the ground state for the system *with* an external static electrical field applied. Delete it since it is useless.
- Calculate a few unoccupied states (`CalculationMode = 3`). The eigenspectrum will be in the file `eigenvalues`. Why don't we find a Rydberg series in the eigenspectrum?
- Repeat the previous calculation with PBE, LB94, and exact exchange. Don't forget to **move** the file `tmp/restart.static` when switching between exchange-correlation functionals.
- Perform a time-dependent evolution (`CalculationMode = 5`), to calculate the optical spectrum of the Na atom. Use a `TDDeltaStrength = 0.05`, polarised in the `x` direction. The multipole moments of the density are output to the file `td.general/multipoles`. You can process this file with the utility `strength-function` to obtain the optical spectrum. If you have computer time to waste, re-run the time-dependent simulation for some other xc choices.

9.0.2 Benzene

Well, the sodium atom is a bit too trivial. Let's try something harder: benzene. you will just need the geometry for benzene to be able to play. Here it is (in Å):

```
C  0.000  1.396  0.000
C  1.209  0.698  0.000
C  1.209 -0.698  0.000
```

```
C  0.000 -1.396  0.000
C -1.209 -0.698  0.000
C -1.209  0.698  0.000
H  0.000  2.479  0.000
H  2.147  1.240  0.000
H  2.147 -1.240  0.000
H  0.000 -2.479  0.000
H -2.147 -1.240  0.000
H -2.147  1.240  0.000
```

Follow now the steps of the previous example. Carbon and Hydrogen have a much harder pseudo-potential than Sodium, so you will probably have to use a tighter mesh. It also takes much more time...

Options Index

S

T

U

V

W

X