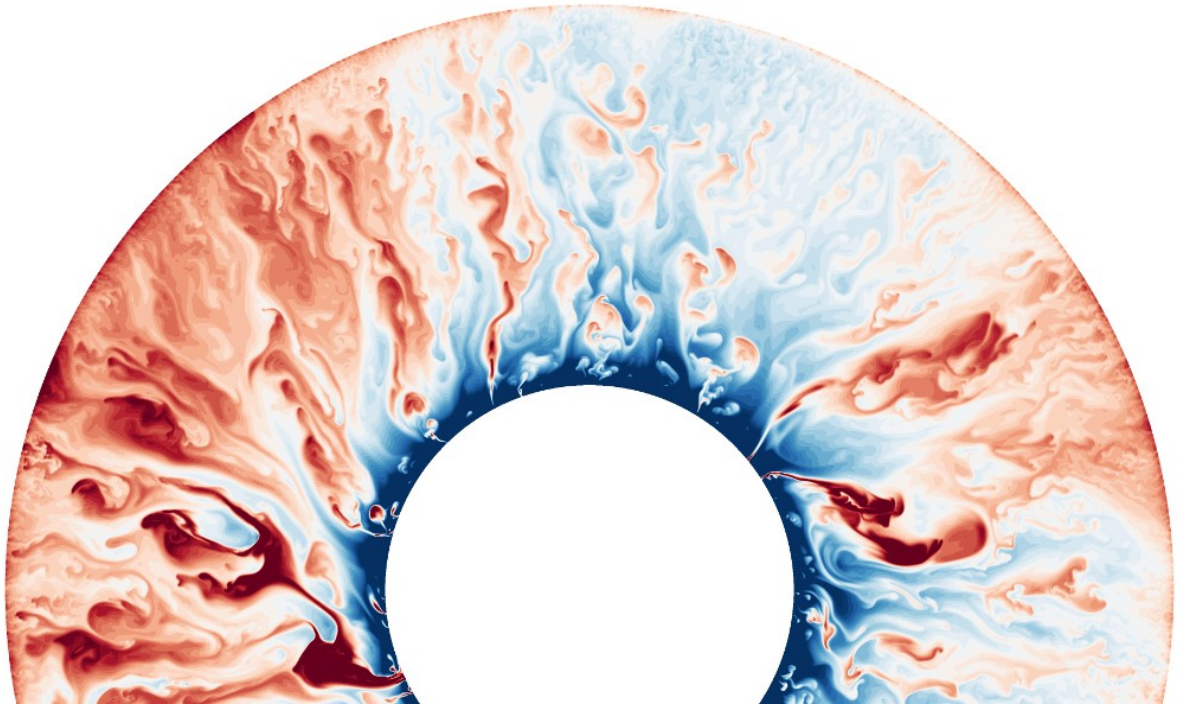
XSHELLS 2.5 : User Manual

Nathanaël Schaeffer
ISTerre/CNRS

October 1, 2020



Chapter 1

Getting started

1.1 Description

XSHELLS is yet another code simulating incompressible fluids in a spherical cavity. In addition to the Navier-Stokes equation with an optional Coriolis force, it can also time-step the coupled induction equation for MHD (with imposed magnetic field or in a dynamo regime), as well as the temperature (and concentration) equation in the Boussinesq framework.

XSHELLS uses finite differences (second order) in the radial direction and spherical harmonic decomposition (pseudo-spectral). The time-stepping uses a highly stable, second order, semi-implicit, predictor-corrector scheme (with only diffusive terms treated implicitly).

XSHELLS is written in C++ and designed for speed. It uses the blazingly fast spherical harmonic transform library SHTns, as well as hybrid parallelization using OpenMP and/or MPI. This allows it to run efficiently on your laptop or on parallel supercomputers. A post-processing program is provided to extract useful data and export fields to python/matplotlib or paraview.

XSHELLS is free software, distributed under the CeCILL Licence (compatible with GNU GPL): everybody is free to use, modify and contribute to the code.

1.2 Transition from versions 1.x to 2.x

Due to the change of time-stepping scheme, in xshells.par:

- adjust time-steps (x3 to x4)
- adjust constants for variable time-step

Due to internal changes, in xshells.hpp:

- some variables have their name changed. see examples.

Slices written by `xspp` are now in Numpy format (instead of text format). As a consequence, matlab/octave display scripts have been dropped.

1.3 Requirements

The following items are *required*:

- a Unix like system (like linux),
- a C++ compiler,
- the FFTW library, or the intel MKL library.
- the SHTns library,

The following items are *recommended*, but not mandatory:

- a C++ compiler with OpenMP 4 support (for task dependencies),
- an MPI library (with thread support),
- the HDF5 library for post-processing and interfacing with paraview,
- Python 3, with NumPy and matplotlib (Python 2.7 should also work),
- a processor supporting the AVX instruction set.
- Gnuplot, for real-time plotting.
- EVTK, for converting slices to VTK format for paraview.

1.4 Installation

FFTW (or the intel MKL) library must now be installed first.

1.4.1 Installing FFTW

FFTW comes already installed on many systems, but in order to get high performance, you should install it yourself, and use the optimization options that correspond to your machine (e.g. `--enable-avx`). Please refer to the FFTW installation guide.

Note that it is possbile to use the intel MKL library instead of FFTW. To do so, you must configure XSHELLS (and SHTns if you configure it manually) with the `--enable-mkl` option.

1.4.2 Configuring XSHELLS

Grab the XSHELLS archive and extract it. Since v2.2.1, the SHTns library may be included. Look for an `shtns` subdirectory. If it is not present, get the latest version of SHTns, and extract it in the `shtns` sub-directory, so that all files from SHTns are in the `shtns` sub-directory (and not in sub-sub-directory).

From the XSHELLS source directory, run:

```
./configure
```

to automatically configure XSHELLS for your machine. This will also automatically configure SHTns and compile it. Type `./configure --help` to see available options, among which `--disable-openmp` and `--disable-mpi`, but also `--enable-knl` to run on the KNL platform (intel compiler needed) or `--enable-mkl` to use the intel MKL library instead of FFTW.

Before compiling, you need to setup the `xshells.hpp` configuration file (see next section). To check that everything works, you may also want to **run the tests**, with `python test.py` or equivalently `make test`.

1.5 Configuration files

There are two configuration files:

- `xshells.hpp` is read by the compiler (compile-time options), and modifying it requires to recompile the program. The corresponding options are detailed in section 2.2.
- `xshells.par` is read by the program at startup (runtime options) and modifying it does not require to recompile the program. This file is detailed in section 2.1.

See chapter 2 for more details. Example configuration files can be found in the `problems` directory.

Before compiling, copy the configuration files that correspond most closely to your problem. For example, the geodynamo benchmark:

```
cp problems/geodynamo/xshells.par .
cp problems/geodynamo/xshells.hpp .
```

and then edit them to adjust the parameters (see sections 2.2 and 2.1).

1.6 Compiling and Running

When you have properly set the `xshells.hpp` and `xshells.par` files, you can compile and run in different flavours:
Parallel execution using OpenMP with as many threads as processors:

```
make xsbig
./xsbig
```

Parallel execution using OpenMP with (e.g.) 4 threads:

```
make xsbig
OMP_NUM_THREADS=4 ./xsbig
```

Parallel execution using MPI with (e.g.) 4 processes:

```
make xsbig_mpi
mpirun -n 4 ./xsbig_mpi
```

Parallel execution using OpenMP and MPI simultaneously (hybrid parallelization), with (e.g.) 2 processes and 4 threads per process:

```
make xsbig_hyb
OMP_NUM_THREADS=4 mpirun -n 2 ./xsbig_hyb
```

Parallel execution using MPI in the radial direction and OpenMP in the angular direction, with (e.g.) 16 processes and 8 threads per process:

```
make xsbig_hyb2
OMP_NUM_THREADS=8 mpirun -n 16 ./xsbig_hyb2
```

Note that `xsbig_hyb2` requires the OpenMP-enabled SHTns (`./configure --enable-openmp`)

Batch schedulers

Some examples for various batch schedulers and super-computers are also available in the `batch` folder.

1.7 Outputs

All output files are suffixed by the job name as file extension, denoted `job` in the following. The various output files are:

- `xshells.par.job` : a copy of the input parameter file `xshells.par`, for future reference.
- `xshells.hpp.job` : a stripped-out version of the file `xshells.hpp` that was used during compilation, for future reference.
- `energy.job` : a record of energies and other custom diagnostics. Each line of this text file is an iteration.
- `fieldX0.job` : the imposed (constant) field X, if any.

- `fieldX_####.job` : the field X at iteration number `####`, if parameter `movie` was set to a non-zero value in `xshells.par`.
- `fieldXavg_####.job` : the field X averaged between previous iteration and iteration number `####`, if parameter `movie` was set to 2 in `xshells.par`.
- `fieldX.job` : the last full backup of field X, or field X at the end of the simulation. Used when restarting a job.

All `field` files are binary format files storing the spherical harmonic coefficients of the field. To produce plots and visualizations, they can be post-processed using the `xspp` program (see chapter 3).

1.8 Limitations and advice for parallel execution

The parallelization of XSHELLS is done by domain decomposition in the radial direction only, using MPI. In addition to this domain decomposition, shared memory parallelism is implemented using OpenMP. There are four variants of the code that differ in the way OpenMP is used:

- `xsbig` uses only OpenMP in the radial direction (no MPI). It can only run on a single node, but does not need an MPI library. This is good for a general purpose desktop or laptop computer, but also on NUMA nodes (although some MPI may lead to better performance). Low memory usage.
- `xsbig_mpi` uses only MPI in the radial direction (no OpenMP). This is good for medium sized problems, running on small clusters.
- `xsbig_hyb` uses both MPI and OpenMP in the radial direction, to reduce the number of MPI processes and memory usage. As a consequence, it is more efficient to use a number of radial shells that is a multiple of the number of computing cores. Beyond 1 thread per radial shell (or in case of imbalance), OpenMP tasks are used to share the work of one shell across several threads, although this leads to a less than ideal scaling.
- `xsbig_hyb2` uses MPI in the radial direction and OpenMP within a radial shell. This strategy usually performs well, allows to efficiently address more computing cores, but uses a lot of memory (an order of magnitude more than `xsbig_hyb`). It is highly recommended to use a number of radial shells that is a multiple of the number of nodes. This mode cannot go beyond 1 MPI process per radial shell, but intra-node shared memory parallelism allows to use all cores of a single node within each process. Note also that the SHTns library compiled with OpenMP is needed.

In any case, **the number of MPI processes cannot exceed the total number of radial shells**. It is often more efficient to use a small number of MPI processes per node (1 to 4) and use OpenMP to have a total number of threads equal to the number of cores.

Because there is no automatic load balancing, some situations where the same amount of work is not required for each radial shell will result in suboptimal scaling when the number of MPI processes is increased. Such situations include (i) solid conducting shells (e.g. a conducting inner core) and (ii) variable spherical harmonic degree truncation (e.g. in a full-sphere problem). In these cases, especially the latter, use pure OpenMP or minimize the number of MPI processes.

Using MPI executables (including hybrid MPI+OpenMP) is thus optimal only if the following conditions are both met:

- all fields span the same radial domain (no conducting solid shells);
- the radial domain does not include the center $r = 0$ (and `XS_VAR_LTR` is not used, see section 2.2.4).

In such cases, XSHELLS should scale very well up to the limit of 1 thread per radial shell, and even beyond with the in-shell OpenMP mode of `xsbig_hyb2` (see scaling example in Figure 1.1).

Advice for various hardwares:

- **Intel servers (Xeon)**: Enabling MPI-3 shared-memory across one socket is recommended. Add `-shared_mem=N` to the command line, where N is the number of cores of each CPU (socket). Using shared-memory across a whole node may help scaling to larger core counts, but significantly reduces raw performance.
- **AMD Epyc Rome**: 4 OpenMP threads always gives the best pereformance. Enable MPI-3 shared-memory across the whole nodes with `-shared_mem=128` for best performance and scaling.
- **Intel KNL**: highly depends on the node configuration.

1.9 Citing

If you use XSHELLS for research work, you should cite the SHTns paper (because the high performance of XSHELLS is mostly due to the blazingly fast spherical harmonic transform provided by SHTns):

N. Schaeffer, *Efficient Spherical Harmonic Transforms aimed at pseudo-spectral numerical simulations*, Geochem. Geophys. Geosyst. **14**, 751-758, doi:10.1002/ggge.20071 (2013)

In addition, depending on the problem you solve, you could cite:

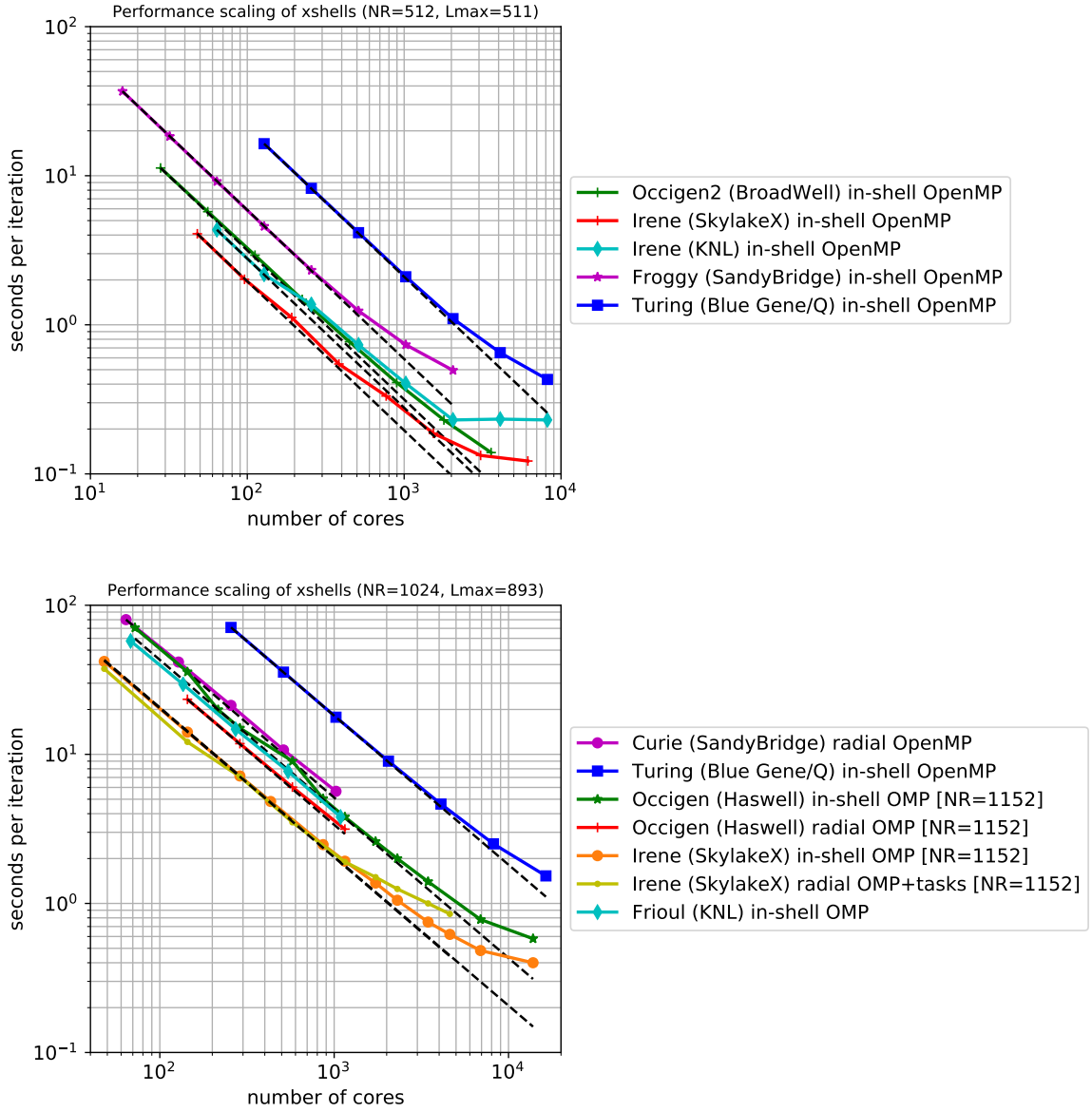


Figure 1.1: Performance scaling of XSHELLS, on french supercomputers with different architectures: *SandyBridge* on *Froggy* (CIMENT) and *Curie* (thin nodes, TGCC); *Haswell* on *Occigen* (CINES); *Blue Gene/Q* on *Turing* (IDRIS); and Knight's Landing (KNL) on *Frioul* (CINES). The ideal scaling for each case is represented by the dashed black lines. Top: geodynamo simulation with $N_r = 512$ radial grid points and spherical harmonics truncated after degree $L_{max} = 511$. Bottom: geodynamo simulation with $N_r = 1024$ or 1152 and $L_{max} = 893$.

- Geodynamo:
  N. Schaeffer et. al, *Turbulent geodynamo simulations: a leap towards Earth's core*, Geophys. J. Int. doi:10.1093/gji/ggx265 (2017)

- Spherical Couette:
  A. Figueroa et. al, *Modes and instabilities in magnetized spherical Couette flow*, J. Fluid Mech. doi:10.1017/jfm.2012.551 (2013)

- Full-spheres: E. Kaplan et. al, *Subcritical thermal convection of liquid metals in a rapidly rotating sphere*, Phys. Res. Lett. doi:10.1103/PhysRevLett.119.094501 (2017)

- Kinematic dynamos:
  N. Schaeffer et. al, *Can Core Flows inferred from Geomagnetic Field Models explain the Earth's Dynamo ?*, Geophys. J. Int. doi:10.1093/gji/ggv488 (2016)

Chapter 2

Setting up the simulation

Example configuration files can be found in the `problems` directory.

2.1 Run-time options: xshells.par

The file `xshells.par` is a simple text file. Each line may contain a single statement like `var = expression`, or a comment starting with `#`. A simple math parser allows to use convenient expressions like `sqrt(4*pi/3)`.

All the following features can be set in `xshells.par`. There is no need to recompile if this file is changed, as it is read and interpreted at program startup.

2.1.1 Equations and controlling parameters

XSHELLS can time-step the Navier-Stokes equation in a rotating reference frame. Optionally it can include (i) a buoyancy force in the Boussinesq approximation, where the buoyancy has one or two sources obeying distinct advection-diffusion equations; and (ii) a Lorentz (or Laplace) force for conducting fluids where the magnetic field obeys the induction equation. Precisely, the following equations can be time-stepped by XSHELLS:

$$\partial_t \mathbf{u} + (2\mathbf{\Omega_0} + \nabla \times \mathbf{u}) \times \mathbf{u} = -\nabla p^* + \nu \nabla^2 \mathbf{u} + (\nabla \times \mathbf{b}) \times \mathbf{b} + (c+T)\nabla \Phi_0 \tag{2.1}$$

$$\partial_t \mathbf{b} = \nabla \times (\mathbf{u} \times \mathbf{b} - \eta \nabla \times \mathbf{b}) \tag{2.2}$$

$$\partial_t T + \mathbf{u}.\nabla(T + T_0) = \kappa \nabla^2 T \tag{2.3}$$

$$\partial_t c + \mathbf{u}.\nabla(c + C_0) = \kappa_c \nabla^2 c \tag{2.4}$$

$$\nabla \mathbf{u} = 0 \tag{2.5}$$

$$\nabla \mathbf{b} = 0 \tag{2.6}$$

where

- $\mathbf{u}$ is the velocity field.

- T and c are respectively the temperature and concentration fields, that contribute to the buoyancy in the Boussinesq formulation.
- $\sqrt{\mu_0 \rho}\,\mathbf{b}$ is the magnetic field (ρ and μ_0 are the fluid density and magnetic permeability, but are not relevant for this equation set).
- ν is the kinematic viscosity of the fluid and is set by the variable `nu` in `xshells.par`.
- $\eta = (\mu_0 \sigma)^{-1}$ is the magnetic diffusivity of the fluid (σ is its conductivity) and is set by the variable `eta` in `xshells.par`. Diffusivity $\eta(r)$ depending on shell radius are also supported (see sec. 2.2.3).
- κ and κ_c are respectively the thermal and chemical diffusivities of the fluid and are set by the variable `kappa` and `kappa_c` in `xshells.par`.
- $\mathbf{\Omega_0}$ is the rotation vector of the reference frame, which is usually along the vertical axis $\mathbf{e_z}$. It is set by the variable `Omega0` in `xshells.par`, while the angle (in radians) between $\mathbf{e_z}$ and $\mathbf{\Omega_0}$ is set by `Omega0_angle` (0 by default). Note that $\mathbf{\Omega_0}$ is always in the $x - z$ plane ($\phi = 0$).
- Φ_0 is the gravity potential (independent of time), controlled by field `phi0` in `xshells.par`.
- T_0 and C_0 are the imposed base temperature and concentration profiles, controlled by fields `tp0` and `c0` in `xshells.par`.
- p^* is the dynamic pressure deviation (from hydrostatic equilibrium), which is eliminated by taking the curl of equation 2.1.

Equation 2.1 (respectively 2.2, 2.3 and 2.4) is time-stepped when `u` (respectively `b`, `tp` and `c`) is set to an initial condition in `xshells.par`. Disabling an equation is as easy as removing or commenting out the corresponding initial condition in `xshells.par`. Note that currently, the **concentration equation cannot be time-stepped without the temperature equation**.

Example If the following lines are found in `xshells.par`:

```
nu = 1.0
eta = sqrt(10)
Omega0 = 2*pi*1e3

u = 0
b = 0
#tp = 0
```

then the viscosity is set to $\nu = 1.0$, the magnetic diffusivity is set to $\eta = \sqrt{10}$, and the rotation rate is set to $\Omega_0 = 2\pi \times 10^3$. The Navier-Stokes (2.1) and the induction equation (2.2) will be time-stepped, but not the temperature and concentration equations (2.3), simulating an isothermal fluid.

Note that it is up to the user to choose dimensional or non-dimensional control parameters. A notable exception is the magnetic field, which is always scaled to the same unit as the velocity field.

MHD without Lorentz force (e.g. kinematic dynamos)

The Lorentz force can be turned off. Just add `no_jxb = 1` in the `xshells.par` file.

Internal representation of vector fields

Vector fields are represented internally using a poloidal/toroidal decomposition:

$$\mathbf{u} = \nabla \times (T\mathbf{r}) + \nabla \times \nabla \times (P\mathbf{r}) \tag{2.7}$$

where $\mathbf{r}$ is the radial position vector, and T and P are the toroidal and poloidal scalars respectively. This decomposition ensures that the vector field $\mathbf{u}$ is divergence-free.

The scalar fields T and P for each radial shell are then decomposed on the basis of spherical harmonics.

2.1.2 Boundary conditions

Magnetic field boundary conditions are that of an electrical insulator outside the computation domain, with or without external sources of magnetic field (see section 2.1.3 for externally imposed magnetic fields).

Temperature or concentration boundary conditions are either fixed value (defined by the T_0 or C_0 profile) or fixed flux (defined by $\partial_r T_0$ or $\partial_r C_0$).

Velocity boundary conditions can be zero, no-slip (with arbitrary prescribed velocity at the boundary) or stress-free.

The inner and outer boundary conditions can be chosen independently. The `BC_U` (for velocity), `BC_T` (for temperature) and `BC_C` (for concentration) entries in `xshells.par` allow to select the appropriate boundary conditions.

> **Example**
> The following lines in `xshells.par` define zero velocity and fixed temperature boundary condition at the inner boundary, and no-slip and fixed flux boundary condition at the outer boudnary.
>
> ```
> BC_U = 0,1    # inner,outer boundary conditions
>               # (0=zero velocity, 1=no-slip, 2=free-slip)
> BC_T = 1,2    # 1=fixed temperature, 2=fixed flux.
> ```

2.1.3 Initial conditions and imposed fields

Predefined fields

Several predefined fields are defined in `xshells_init.cpp`. The command `./list_fields` prints a list of these predefined fields, with their name in the first column. You can simply use this name in the `xshells.par` file to define an initial condition. You can also add your own by editing `xshells_init.cpp`.

Imposed fields are only supported for the gravity potential `phi0` and for the basic state of temperature `tp0` and concentration `c0`. Imposed magnetic fields can be obtained through the appropriate boundary conditions (magnetic fields generated by currents outside the computation domain only). Some predefined magnetic field include these boundary conditions, making them actually *imposed* fields (and are labeled as such). Note also that a linear mode exist which support arbitrary base fields (see §2.2.7).

Example The following lines in `xshells.par` set up the geodyanmo benchmark initial conditions.

```
E = 1e-3
Pm = 5
Ra = 100

u = 0                  # initial velocity field
b = bench2001*5/sqrt(Pm*E)  # initial magnetic field (scaled by
                            #   sqrt(1/(Pm*E)))
tp = bench2001*0.1     # initial temperature field
tp0 = delta*-1         # imposed (base) temperature field
phi0 = radial*Ra/E     # radial gravity field (multiplied by
                       #  Ra/E to match geodynamo benchmark)
```

Field files as initial conditions

In addition, any `field` file can be given as initial condition. If the radial grid is not the same, the field must be interpolated on the new grid. To avoid mistakes, interpolation is disabled by default and must be enabled by `interp = 1` (often found near the end of the `xshells.par` file).

Example The following lines start from the velocity field saved in file `fieldU.previous_job`, which was performed at different parameters and with a different number of radial grid points.

```
u = fieldU.previous_job   # initial velocity field
interp = 1     # allow interpolation, to be able to use fields
       # defined on a different radial grid as initial condition.
```

2.1.4 Forcing

Besides thermal convection, mechanical forcing can be imposed at the boundaries.

Predefined variable `a_forcing` and `w_forcing` define the amplitude and frequency of a forcing. The precise nature of the forcing (e.g. differential rotation) must be defined in the `xshells.hpp` file before compilation (see section 2.2.6).

2.1.5 Spatial discretization

Radial grid

XSHELLS uses second order finite differences in radius. The total number of radial grid points is defined in `xshells.par` by the variable `NR`. The radial extent of each field is set using the corresponding `R_X` variable, which stores a pair of increasing positive real numbers defining the radial extent of the field. The `NR` grid points will be distributed between radii corresponding to the minimum and maximum of these values. Currently, only the magnetic field can extend beyond the velocity field, modeling conducting solid layers.

> **Example** The following lines in `xshells.par` define the radial extent of the fields:
>
> ```
> R_U = 7/13 : 20/13
> R_B = 0.0 : 20/13
> R_T = 7/13 : 20/13
> ```

The default grid refines the number of points in the boundary layers, and this refinement can be controlled by the variable `N_BL` that stores a pair of integers, the first and second being the number of points reserved for the inner and outer boundary layer respectively, reinforcing the normal refinement. The code generating the grid can be found in the `grid.cpp` file.

> **Example** The following lines in `xshells.par` define a grid with a total of 100 radial grid points, with 10 and 5 points reserved to the refinement of the inner and outer boundary layer respectively.
>
> ```
> NR = 100
> N_BL = 10,5
> ```

Alternatively, a radial grid can be loaded:

- from a text file containing the radius of each grid point (increasing) in a separate line.
- from a previously saved field (see section 1.7).

In both cases, simply indicate the filename in the `r` variable. It will override the `NR` and `N_BL` variables.

Example The following line in `xshells.par` will use the same grid as the field stored in file `fieldU_0001.previous`

```
r = fieldU_0001.previous      # load grid from file
```

Angular grid and spherical harmonic truncation

XSHELLS uses spherical harmonics to represent fields on the sphere:

$$f(\theta,\phi) = \sum_{m=0}^{M} \sum_{\ell=mK}^{L} f_\ell^{mK} Y_\ell^{mK}(\theta,\phi) \tag{2.8}$$

where Y_ℓ^m is the spherical harmonic of degree ℓ and order m. The expansion uses a K-fold symmetry in longitude (ϕ) and is truncated at maximum degree L and order MK. If $K = 1$ and $M = L$, it is the standard triangular truncation.

L, M and K are set in `xshells.par` using the `Lmax`, `Mmax` and `Mres` variables respectively. You must ensure that $L \geq MK$.

The angular grid (spanning the co-latitude θ and longitude ϕ) consists of `Nphi` regularly spaced points in longitude, and `Nlat` gauss nodes in latitude. If these are not specified, XSHELLS will choose the values for `Nlat` and `Nphi` in order to ensure best performance and no aliasing of modes (`Nlat` $> 3L/2$ and `Nphi` $> 3M$).

Example These lines limit the spherical harmonic degree to 170. A 3-fold symmetry is used, and the maximum harmonic order is $56 \times 3 = 168$.

```
Lmax = 170   # max degree of spherical harmonics
Mmax = 56    # max fourier mode (phi)
Mres = 3     # phi-periodicity.
```

Most likely, 180 regularly spaced points in longitude and 256 gauss nodes in latitude will be used here.

2.1.6 Time-stepping

XSHELLS uses semi-implicit (or IMEX) time steppers, where the diffusive terms are treated implicitly, while all other terms (including non-linear terms) are treated explicitly. Since v2.4, XSHELLS offers a selection of time steppers:

- PC2 (the default) is a second order predictor-corrector scheme. Although this scheme requires three times more work per time-step than the classical Crank-Nicolson-Adams-Bashforth (CNAB) scheme, it allows time-steps from three to four times that of CNAB, leading to shorter time-to-solution[1].

[1] before version 2.0, a classical CNAB scheme was used

- CNAB2 is the classical second order Crank-Nicolson-Adams-Bashforth (CNAB) scheme[2].
- SBDF2 is the semi-implicit Backward Difference Formula of order 2. This scheme[3] has a stringent time-step restriction with the Coriolis force.
- SBDF3 is the semi-implicit Backward Difference Formula of third order[4].
- BPR353 is a third order Runge-Kutta scheme featuring good stability and high accuracy[5].

Simply add `stepper = SBDF3` in the `xshells.par` file to use the SBDF3 scheme. Note that the allowable time steps can vary largely between these schemes. The automatic time-step selection (see below) should handle most problems well.

Both PC2 and CNAB2 use the Crank-Nicolson scheme for the implicit part, and as a consequence are not very accurate when computing decay rates (e.g. magnetic dipole decay rate). On the other hand SBDF2 and SBDF3 are accurate for decay rates, but non-linear terms like advection impose small time-steps for the scheme to remain stable.

The time-step of the numerical integration is set by `dt` in `xshells.par`.

The `sub_iter` variable is half the number of time-steps taken before any diagnostic is computed and written to file `energy.job` or displayed on screen. For example, if `sub_iter = 50`, then 100 time-steps will be performed before computing and printing some diagnostics. This is then called an iteration.

The `iter_max` variable is the total number of iterations, so that the total number of time-steps before the code will stop is `iter_max` $\times 2 \times$ `sub_iter`.

By setting `dt_adjust = 1`, an automatic time-step adjustment can be turned on. The time-step adjustment can be controlled through advanced options (see §2.1.10). In that case, the number of sub-iterations `sub_iter` is also adjusted so that an iteration is a constant time span, and thus the outputs happen at fixed time intervals $\Delta T = 2 \times$ `sub_iter` $\times$ `dt`.

Finally, `iter_save` controls the number of iterations before a (partial) snapshot is saved to disk.

[2]CNAB2: see eq. 2.9 of Wang & Ruuth (2008) https://www.jstor.org/stable/43693484.
[3]SBDF2: see eq. 2.8 of Wang & Ruuth (2008) https://www.jstor.org/stable/43693484.
[4]SBDF3: see eq. 2.14 of Wang & Ruuth (2008) https://www.jstor.org/stable/43693484.
[5]BPR353: see page A49 of Boscarino, Pareschi, Russo (2013) https://doi.org/10.1137/110842855 https://arxiv.org/abs/1110.4375.

Example The following lines in `xshells.par` will use a time-step of 0.01 for the numerical integration:

```
dt_adjust = 0     # 0: dt fixed (default), 1: variable time-step
dt = 0.01         # time step
iter_max = 300    # iteration number (total number of text and
                  #   energy file ouputs)
sub_iter = 25     # sub-iterations (the time between outputs
                  #   = 2*dt*sub_iter is fixed even with variable dt)
iter_save = 10    # number of iterations between field writes
```

Output will occur every $\Delta T = 0.01 \times 25 \times 2 = 0.5$ time units (an iteration). The program will stop after `iter_max`=300 outputs (or iterations), spanning a total physical time of $t_{end} - t_{start} = 150.0$. Partial fields are saved every 10 iterations, or every 5.0 physical time units, if `movie = 1` is set (see below).

2.1.7 Real time plotting

At each iteration, XSHELLS can plot the kinetic and magnetic energies as a function of time, using gnuplot. Note that the plots are refreshed every iteration, but no more than once every two seconds. This allows to follow program execution in real-time, but might not be useful for high performance distributed jobs. The interaction with gnuplot must be turned on by passing the `--enable-gnuplot` option to `./configure`.

The variable `plot` in the file `xshells.par` then allows some control:

- `plot = 0`: disables plotting.
- `plot = 1`: shows plot on display; if no display found, write to png file instead. This is the default.
- `plot = 2`: saves plot to png file only.
- `plot = 3`: shows plot on display (if available) and also saves plot to png file.

2.1.8 Time lapse field snapshots

The parameter `movie` controls the field snapshots, saved every `iter_save` iterations (see above).

- `movie = 1` : the initial field is saved to `fieldX_0000.job`, after `iter_save` iterations the fields are saved to `fieldX_0001.job`, then `fieldX_0002.job`, and so on.
- `movie = 0` : no such fields are saved.

- `movie = 2` : in addition to the snapshots of the fields, the time-average since the last snapshot is also computed and saved.

The parameter `prec_out` controls the precision (single or double precision) of the snapshot files. The snapshots are saved in double precision by default (`prec_out = 2`), which allows restarting or computing gradients. If you need to save disk space, you can use single precision instead by setting set `prec_out = 1`, thus writing field files of half size. To save further disk space, snapshots can be truncated at lower spherical harmonic degree and order, using the parameters `lmax_out` and `mmax_out`, respectively.

Sometimes, single precision is not enough but double precision is not needed. Setting set `prec_out = 3` uses the custom fp48 format which takes 25% less space than double precision, while keeping high accuracy.

The snapshots can then be post-processed with `xspp` to produce plots or movies (see 3). Note that field in fp48 format are read seamlessly by `xspp` which can also convert between formats. However, `pyxshells` cannot read the fp48 format.

> **Example** The following lines in `xshells.par` instruct the program to output snapshots and time-averages of the axisymmetric component of the fields, every `iter_save` iterations:
>
> ```
> movie = 2       # 0=field output at the end only (default),
>                 # 1=output every iter_save, 2=also writes
>                 # time-averaged fields
> lmax_out = -1   # lmax for movie output (-1 = same as Lmax, which
>                 #  is also the default)
> mmax_out = 0    # mmax for movie output (-1 = same as Mmax, which
>                 #  is also the default)
> prec_out = 2    # write double precision snapshots.
> ```

2.1.9 Checkpointing and restarting capabilities

By default after initialization the job starts at the beginning (iteration 0). It is easy to start a new job by using as input fields the `field` files written by a previous job, effectively continuing that job.

Sometimes, it is useful to automatically continue a stopped or killed job (e.g. in batch execution environments found in high-performance computing machines). By default, a full resolution snapshot is written to disk every four hours. Parameters found in `xshells.par` allow to tune that interval, and enable restart from these checkpoint automatically when the program is run again.

For increased safety, when writing a new checkpoint (or backup) to file `fieldX.jobname`, the previous one is first renamed to `fieldX_back.jobname`. This file may allow to continue

a simulation in case of an unexpected termination of the program while writing the new checkpoint.

If `restart = 1`, the program will start by looking in the current directory for checkpoint files that have been saved by a previous run with the same job name, and use these to resume that job.

> **Example**
> Suppose that on a supercomputer, the wall time of the jobs is limited to 24 hours. In order to run a job that spans several days, the following lines in `xshells.par` allow a job to be resumed by simply resubmitting it:
>
> ```
> restart = 1         # 1: try to restart from a previous run with
>                     #   same name. 0: no auto-restart (default).
> backup_time = 470   # ensures that full fields are saved to disk at
>                     #   least every backup_time minutes, for restart.
> nbackup = 3         # number of backups before terminating program
>                     # (useful for time-limited jobs).
>                     #  0 = no limit (default)
> ```
>
> In addition, a checkpoint (or backup) is written to disk every 470 minutes, and the program will stop after writing the third backup, thus leaving 30 minutes of safety time for program initialization and file writing time. In case of a system failure, no more than 470 minutes of computing time will be lost.

2.1.10 Advanced options

Fine tuning of the automatic time-step selection is possible through some variables.

`C_u` is a safety factor for the standard CFL (based on the velocity and the grid size), which depends on the time-scheme. For the PC2 time-scheme used by xshells, the default value is `C_u = 0.65`. You may need a more stringent value for a given problem, but if your simulation still does not work for a value lower than 0.1, the problem is probably elsewhere. `C_alfv` control the time-step adjustment (active if `dt_adjust=1`), regarding Alfvén criteria (due to magnetic field). The lower the values of `C_u` and `C_alfv`, the smaller the adjusted time step will be.

In addition, to prevent too many time step adjustments, if `dt_tol_lo < dt/dt_target < dt_tol_hi`, no time-step adjustment is done.

Example

```
C_u = 0.65          # default: 0.65
C_cori = 1          # default: 1
C_alfv = 2.8        # default: 2.8
dt_tol_lo = 0.9     # default: 0.9
dt_tol_hi = 1.05    # default: 1.05
```

Variable spherical harmonic degree truncation is controlled by setting the variable `rsat_ltr` in `xshells.par`. This is possible only if XSHELLS was compiled with variable truncation enabled (see section 2.2.4).

Spectral convergence of fields is checked by comparing the maximum of the end of the spectrum (four last modes) with the maximum of the spectrum. The first few modes are ignored (because their amplitude can be related directly to forcing or imposed fields). The variables `sconv_lmin` and `sconv_mmin` that can be set in `xshells.par` define the first spherical harmonic degree and order respectively that are considered when checking spectral convergence in the ℓ or m spectra. The default value is 3 for both variables. Setting a value larger than `Lmax` disables spectral convergence checks. Note also that the convergence checks are skipped when only a few modes are computed.

In addition, the maximum allowed value for Sconv can be set using variable `sconv_stop` (default value is 0.3). When the computed Sconv is larger than this limit, the program stops. Setting `sconv_stop = 0` disables this safety check.

Example

```
sconv_lmin = 3     # ignore l=0-2 for spectral convergence.
sconv_mmin = 1     # ignore m=0 for spectral convergence.
sconv_stop = 0.3   # set the maximum value of Sconv before the program stops.
```

The SHTns library can be controlled in terms of algorithm used for transforms, and in terms of polar optimization threshold.

The `sht_type` variable allows to constrain the transform method used:

- `sht_type = 0` : select fastest method using a classic Gauss-Legendre grid (default setting).
- `sht_type = 1` : select fastest method, allowing also regular grids (with DCT) which may be faster for small `Mmax`.

- `sht_type = 2` : impose a regularly spaced grid (not recommended as it is often slower).
- `sht_type = 3` : force a regularly spaced grid using DCT (not recommended as it is often slower).
- `sht_type = 4` : debug mode; initialization time is reduced, but a default method is used (no selection of fastest method).
- `sht_type = 6` : use a Gauss-Legendre grid with on-the-fly computation (preferred when parallel execution or big resolutions).

Finally, the polar optimization threshold can be adjusted with `sht_polar_opt_max`, the value below which coefficients near the pole are neglected. To give the reader some more insight, here are a some possible values and their impact:

- `sht_polar_opt_max = 0` : no polar optimization.
- `sht_polar_opt_max = 1e-14` : very safe optimization (default).
- `sht_polar_opt_max = 1e-10` : safe optimization.
- `sht_polar_opt_max = 1e-6` : aggressive optimization.

OpenMP tasks with dependencies are used by `xsbig_hyb` whenever the number of shells cannot be distributed evenly enough between the available threads. However, OpenMP tasks with dependencies can cause problems with some compilers (e.g intel compilers prior to version 18), and can be turned off by setting `omp_tasks = -1`, or forced with `omp_tasks = 1` in the `xshells.par` file.

2.1.11 Beta features

The following features have not been thoroughly tested and may not work flawlessly in all situations. If you want to use them, please test and report bugs.

Non-linear terms in linear mode can be included to compute the saturation of an instability growing on a base flow. The program must be compiled in linear mode (see 2.2.7), and each non-linear term can be activated separately using a comma separated list in `xshells.par`:

```
nonlin = ugu,uxb,jxb,ugt,ugc
```

where `ugu`, `uxb`, `jxb`, `ugt` and `ugc` activate respectively $\mathbf{u}.\nabla\mathbf{u}$, $\nabla\times(\mathbf{u}\times\mathbf{b})$, $(\nabla\times\mathbf{b})\times\mathbf{b}$, $\mathbf{u}\nabla T$, $\mathbf{u}\nabla C$ for the perturbations $\mathbf{u}$,$\mathbf{b}$,T and C in the equations (see 2.1.1).

Example To compute a kinematic dynamo growing on a saturated hydrodynamic instability, use:

```
nonlin = ugu,uxb      # include only u.grad(u) and curl(u x b)
```

2.2 Compile-time settings: xshells.hpp

All the following settings can be found in `xshells.hpp`. You have to recompile the program if you change this file.

2.2.1 Custom diagnostics

Enable by uncommenting:

```
#define XS_CUSTOM_DIAGNOSTICS
```

In addition to the total energy of the three fields U, B and T, which are saved every 2 `sub_iter` time steps (see section 2.1.6), custom diagnostics can be defined in the `custom_diagnostic()` function, found in the `xshells.hpp` file. They are computed every iteration and saved in `energy.job` after the energies. The best is to look at the existing diagnostics defined in the `custom_diagnostic()` function to add your own.

2.2.2 Variable time-step

Enable compilation of variable time-step code by uncommenting:

```
#define XS_ADJUST_DT
```

In addition,variable time-step must also be allowed by setting `dt_adjust = 1` in file `xshells.par` (see also section 2.1.6).

2.2.3 Variable conductivity

In equation 2.2, conductivity can depend on radius r. To define a conductivity profile $\eta(r)$, uncomment:

```
#define XS_ETA_PROFILE
```

and then define your profile in the `calc_eta()` function, found in the `xshells.hpp` file. The purpose of `calc_eta()` is simply to fill the array `etar` with values of the magnetic diffusivity for every radial shell. The program handles continuous profiles as well as discontinuities in $\eta(r)$ properly and automatically.

2.2.4 Variable spherical harmonic degree truncation

In order to compute in a full sphere and avoid problems near $r = 0$, the spherical harmonic expansion must be truncated at low degree near $r = 0$. XSHELLS can truncate the spherical harmonic expansions at a different degree for each shell, when the following line is uncommented in `xshells.hpp`:

```
#define VAR_LTR 0.5
```

The value of `VAR_LTR` (0.5 in the line above, which is a good choice for full-sphere computations) is used as α in the formula to determine the truncation degree ℓ_{tr}:

$$\ell_{tr}(r) = L_{max} \sqrt{\frac{r}{\alpha r_{max}}} + 1$$

where L_{max} is defined by `Lmax` in `xshells.par` and r_{max} is the radius of the last shell. Note that ℓ_{tr} cannot exceed L_{max}.

Note also that α can be overridden by the `rsat_ltr` variable in `xshells.par`.

2.2.5 Hyper-diffusivities

To avoid computing small-scales that may not be dynamically relevant, the easy way is to employ hyper-diffusivity, which increase the diffusivity of small-scales. XSHELLS can be compiled with hyper-diffusivities when the following line is uncommented in `xshells.hpp`:

```
#define XS_HYPER_DIFF
```

The formulae used to compute the viscosity is the one proposed by Nataf & Schaeffer (2015):

$$\nu(\ell) = \begin{cases} \nu_0 & \text{for } l \le \ell_c \\ \nu_0\, q^{\ell-\ell_c} & \text{for } \ell > \ell_c \end{cases} \tag{2.9}$$

with $q = (\nu_{max}/\nu_0)^{1/(L_{max}-\ell_c)}$. The cutoff ℓ_c and the ratio ν_{max}/ν_0 are set respectively by the `hyper_diff_l0` and `hyper_nu` parameters in `xshells.par`. If `hyper_diff_l0 = 0`, hyper-diffusivity is disabled. Similarly, `hyper_nu = 1` gives uniform viscosity, while `hyper_nu = 100` leads to a viscosity at $\ell = L_{max}$ that is 100 times larger than at large scales.

Note that thermal and magnetic diffusivities can also be treated similarly, using respectively the `hyper_kappa` and `hyper_eta` parameters. The cutoff parameter ℓ_c is the same for all diffusivities.

2.2.6 Boundary forcing

Amplitude and frequency are set at runtime by `a_forcing` and `w_forcing` in the `xshells.par` file.

Time dependent boundary forcing are defined in the function `calc_Uforcing()`, found in the `xshells.hpp` file. In this function you must define a name for your forcing through the macro `U_FORCING`. The angular velocity of the solid bodies (defining the boundary of the fluid shell) can be set in this function. It will be used as a boundary condition for the flow if no-slip boundaries are used (see section 2.1.2).

See the example found in the `problems/couette/` folder for more details, and uncomment the part of the function corresponding to your problem. In particular axial differential rotation of the inner or outer boundary can be used to simulate a spherical Couette flow; equatorial differential rotation (or spin-over) can be used to simulate precession or nutation.

Note that the rotation rate of the solid bodies is also used in the induction equation if the magnetic field extends into the solids (conducting solid shells).

Arbitrary stationary boundary flows You can impose arbitrary stationary flows at the solid boundaries. Uncomment:

```
#define XS_SET_BC
```

and change the `set_U_bc()` function found in `xshells.hpp` according to your needs. Note that the boundary conditions for the poloidal velocity field is stored in the shell beyond the first or last fluid shells (respectively `NG-1` and `NM+1`). See for example the `xshells.hpp` file in the `problems/full_sphere/` folder. Note that the solid should not be conducting if this feature is used, as no corresponding solid flow will be generated.

2.2.7 Linearized equations and base fields

To replace the equations with their linearized version (no $(\nabla \times \mathbf{u}) \times \mathbf{u}$, no $(\nabla \times \mathbf{b}) \times \mathbf{b}$, no $\mathbf{u}.\nabla c$), uncomment:

```
#define XS_LINEAR
```

Note that spherical harmonic transforms are not needed anymore if there are no base fields, leading to a much faster program. Base fields are also supported, and can be set using `u0`, `b0` and `tp0` in the `xshells.par` file. See also the example located in `problems/taw`.

2.2.8 Mean-field dynamo

To include an axisymmetric alpha effect in the induction equation, add the following in the `xshells.hpp` file:

```
#define XS_MEAN_FIELD
```

And provide the alpha field using the `init_alpha` function. See the example located in `problems/kinematic_dynamos`.

Chapter 3

Post-processing and visualization

Several tools are available for post-processing and visualization of data produced by the XSHELLS code:

- `xsplot.py`: a python module for plotting 1D or 2D data.
- `xsplot`: command-line interface to xsplot.py.
- `xspp`: command-line tool for extracting slices, profiles, spectra and more from the field files.
- `pyxshells.py`: a python module that can load and handle the binary XSHELLS field files.

The python module `xsplot` is provided to load and display data from XSHELLS. It can be used interactively or within scripts. Such Python scripts using matplotlib and xsplot are located in the `matplotlib` dierectory, and can be called from command line.

`xsplot` can also be used directly from command line and it will guess the type of file and display it accordingly.

The python modules should be installed by calling `make install-py`, or explicitly `python setup.py install --user` from the `python` subdirectory. In addition, it is convenient to copy the small script `python/xsplot` in your path, so that you can invoke xsplot from any directory.

3.1 Plotting time-dependent quantities

Time-dependent quantities, such as the energies and other custom diagnostics which are stored in the `energy.job` file (see section 2.2.1) are also easy to plot with the `xsplot` tool.

Example By default, xsplot displays kinetic and magnetic energy as a function of time:

```
xsplot energy.job
```

A list of all available quantities is also shown.
To plot other quantities (see section 2.2.1), for instance kinetic energy and viscous dissipation, use:

```
xsplot energy.job -c Eu,D_nu
```

The `load_diags` function in the `xsplot` module can be used to retrieve conveniently any diagnostic. It returns a dictionary of time series for easy plotting.

Example To plot the magnetic to kinetic energy ratio as a function of time:

```
> from pylab import *
> import xsplot
> d = xsplot.load_diags('energy.job')
> t = d['t']
> plot(t, d['Eb']/d['Eu'])   # plot magnetic to kinetic energy ratio
```

3.2 Dealing with 3D fields

Fields are stored in binary files (see 1.7), using a custom format. They can be handled after the simulation by the `xspp` command line program. Alternatively, the `pyxshells` python module can also read, write and handle these files (see section 3.2.5).

3.2.1 Using the xspp command-line tool

Compile the program by typing `make xspp`. Invoking it without arguments (by running `./xspp`) will print a help screen including the commands and their syntax.

Example The following will display information about the file `fieldU.job` (resolution, precision, time of the snapshot, ...):

```
./xspp fieldU.job
```

To compute the energy and maximum value of the curl of the field:

```
./xspp fieldU.job curl nrj max
```

To extract the field values along a line spanning the x-axis from $x = -1$ to $x = 0.8$, and also display total energy of field:

```
./xspp fieldU.job line -1,0,0 0.8,0,0 nrj
```

Add two fields and save the result to a new file (the first file will set the resolution for the result):

```
./xspp fieldT_0004.job + fieldT0.job save fieldT_total_0004.job
```

Extract only a given range of spherical harmonic coefficients (2 to 31) and computes the corresponding energy:

```
./xspp fieldB.job llim 2:31 nrj
```

Note that `xspp` is not parallelized using MPI, so that for very big cases you might run out of memory (although it can operate out-of-core – without actually loading the whole file in memory – in some cases). As a workaround you can always reduce the spherical harmonic truncation while reading your big files with the `llim` option (see example avobe).

3.2.2 Extract 2D slices

One of the most common usage for `xspp` is to extract two-dimensional slices of the 3D data stored in spectral representation in the field files. Four types of 2D slices are available:

- Meridian cuts (a plane containing the z-axis), with the `merid` command;
- Equatorial cuts (the plane $z = 0$), with the `equat` command;
- Surface data (on a sphere of given radius r), with the `surf` command;
- Disc cuts (an arbitrary plane), with the `disc` command;

When these commands are given to `xspp`, NumPy files corresponding to the required cuts are written to the current directory. These NumPy files (`*.npy`) can then be loaded and displayed using Python with NumPy and matplotlib (see next section).

Example A meridian cut at $\phi = 0$:

```
./xspp fieldU.job merid
```

An equatorial cut, and a meridian cut at $\phi = 45$degrees, of the vorticity (curl of U)

```
./xspp fieldU.job curl equat merid 45
```

Extract the field at the spherical surface closest to $r = 0.9$, using only the symmetric components.

```
./xspp fieldU.job sym 0 surf 0.9
```

Make a cut at $z = 0.7$, using 200 azimuthal points, with field truncated at harmonic degree 60:

```
./xspp fieldU.job llim 0:60 disc 200 0,0,0.7
```

3.2.3 plotting with python/matplotlib

The python module `xsplot` is provided to load and display cuts produced by `xspp`. It can be used interactively or within scripts. Such Python scripts using matplotlib and xsplot are located in the `matplotlib` dierectory, and can be called from command line. `xsplot` can also be used directly from command line and will guess the type of cut of your file and display it accordingly.

The python module should be installed by calling `make install-py`, or explicitly `python setup.py install --user` from the `python` subdirectory. In addition, it is convenient to copy the small script `python/xsplot` in your path, so that you can invoke xsplot from any directory.

Example Produce a meridian and an equatorial cut with `xspp`:

```
./xspp fieldU.job merid equat
```

From command prompt, quickly load and plot all components of the field in this meridional slice, as well as in the equatorial plane.

```
xsplot o_merid_0.npy o_equat.npy
```

`xsplot` has several convenient options. For instance you can plot the first two components only (using `-c` option), and specify a range for the colormap (using the `-z` option):

```
xsplot o_equat_0.npy -c 0,1 -z "(-1e-3,2e3)"
```

Alternatively, from an Ipython interpreter (or notebook, or script), load and plot the ϕ-component of the field in the meridional and equatorial slices:

```
> import xsplot
> a = xsplot.load_slice_npy('o_merid_0.npy')
> xsplot.plot_slice(a, 2)    # plot third (phi) component
> d = xsplot.load_slice_npy('o_equat.npy')
> xsplot.plot_slice(d, 1)    # plot second (phi) component
```

Several useful scripts for basic and advanced plotting can be found in the `matplotlib` directory.

3.2.4 3D visualization with paraview

From the paraview website: *ParaView is an open-source, multi-platform data analysis and visualization application. ParaView users can quickly build visualizations to analyze their data using qualitative and quantitative techniques.*

Full spatial fields can be saved to XDMF format, which can be loaded by paraview. Note that the HDF5 library is required for this to work, and must be found by the `configure` script. If so, Simply run:

```
make xspp
./xspp fieldB_0004.job hdf5 B_cartesian.h5
```

The file `B_cartesian.h5.xdmf` describes the cartesian components of the vector field B on a spherical grid that can be read directly by paraview (if prompted for a loader, select 'XDMF').

3.2.5 Advanced post-processing using pyxshells

For more complex post-processing, xspp may not be enough. The python module `pyxshells` allows you to quickly write your own scripts to work directly with the spectral fields stored in the `field` files output by XSHELLS, cast them to spatial domain, and so on.

To install the python modules, type `make install-py`. You can then import the `pyxshells` module from any python script. Note that the `pyxshells` module requires the `shtns` python module, which can be installed from the shtns directory using:

```
./configure --enable-python --disable-openmp
make
python setup.py install --user
```

> **Example** Here is an example of what pyxshells can do:
>
> ```
> > import pyxshells
> > f = pyxshells.load_field('fieldU.job')
> > r = f.grid.r    # the radial grid
> > KE = f.energy()  # computes the energy
> > f.sht.set_grid()   # prepare the spherical grid
> > u3D = f.spat_full() # the 3D field in spherical coordinates
> > ur,ut,up = f.spat_shell(len(f.grid.r)-1)   # surface field
> > ur,ut,up = f.spat_merid()   # meridional slice
> > ur,ut,up = f.spat_equat()   # equatorial slice
> ```

Chapter 4

Hacking

4.1 Main source files

The main programs (`xsbig`, `xsbig_mpi` and `xsbig_hyb`) all share the same source code:

- `xshells_big.cpp` is the main source file, including the main loop, and all the solver logic.
- `xshells.hpp` is where a lot of customization goes on. See section 2.2.
- `grid.cpp` contains functions related to the radial grid and to the banded matrix linear solver.
- `xshells_spectral.cpp` contains the definition of the classes used to describe spectral fields (scalar and vector), and the implementation of most associated methods.
- `xshells_io.cpp` contains methods and functions to load and store fields to file on disk.
- `xshells_physics.cpp` generation of evolution matrices and computation of physical quantities.
- `xshells_init.cpp` initialization functions and predefined fields.
- `findiff.cpp` finite difference derivatives.
- `xshells_linop.cpp` Linear operators (banded matrices) and solvers.
- `xshells_stepper.cpp` Support of various time-steppers.
- `xshells_sparse.cpp` To handle the Coriolis force implicitely, we rely to large sparse matrices.

The post-processing program `xspp` uses the previous source files but also:

- `xspp.cpp` as main source file (instead of `xshells_big.cpp`)
- `xshells_spatial.cpp` contains the definition of the classes used to describe spatial fields (scalar and vector), their relationship with spectral representation and associated methods.
- `xshells_render.cpp` contains method implementations for rendering fields on grids and slices, as well as output to hdf5 files.

4.2 Doxygen

The source code also contains Doxygen documentation comments. Run `make docs` to generate the documentation targeted at developers and contributors in the `doc/html/` folder.

4.3 Mercurial repository

To track the changes to the code, the distributed version control system Mercurial is used. The main mercurial repository, found at `https://bitbucket.org/nschaeff/xshells` allows you to use the latest (unstable) revision (at your own risk). You can also fork it and propose to merge your changes.

Chapter 5

Frequently Asked Questions

Why is XSHELLS so fast? Short answer: it uses SHTns for spherical harmonic transforms and tries to preserve data locality. A Longer answer can be found in this presentation: http://dx.doi.org/10.6084/m9.figshare.1304532.

What are the differences with the PARODY code? The numerical methods are basically the same, but their implementations are different. The PARODY code is written in Fortran. The performance and scalability of XSHELLS are better.

Why is XSHELLS not written in Fortran? Because we don't like Fortran, and we would not be able to get the same level of performance out of a Fortran code. But maybe you could !