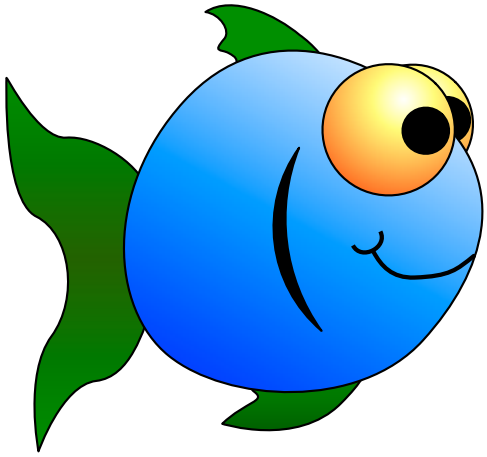


XMDS2 Documentation

Release 2.1.2

Graham Dennis, Joe Hope and Mattias Johnsson

October 19, 2012

CONTENTS

This website provides the documentation for XMDS2, a software package that allows the fast and easy solution of sets of ordinary, partial and stochastic differential equations, using a variety of efficient numerical algorithms.

If you publish work that has involved XMDS2, please cite it as Comput. Phys. Commun. 184, 201-208 (2013).

CHAPTER

ONE

INTRODUCTION

Welcome to **XMDS2** (codenamed *xpdeint*), which is an all-new version of **XMDS**. Prepare for fast, easily-extended simulations with minimal code error.

The purpose of XMDS2 is to simplify the process of creating simulations that solve systems of initial-value first-order partial and ordinary differential equations. Instead of going through the error-prone process of writing by hand thousands of lines of code, XMDS2 enables many problems to be described in a simple XML format. From this XML description XMDS2 writes a C++ simulation that solves the problem using fast algorithms. Anecdotally, the code generated by XMDS2 is as fast as, or faster than, code hand-written by an expert, but by using XMDS2 the time taken to produce the simulation is significantly reduced.

XMDS2 can be used to simulate almost any set of (coupled) (partial) (stochastic) differential equations in any number of dimensions. It can input and output data in a range of data formats, produce programs that can take command-line arguments, and produce parallelised code suitable for either modern computer architectures or distributed clusters.

If this is your first time with XMDS, then an ideal place to start is the *Quickstart Tutorial*, where we will show you how to write a basic simulation. *Installation* instructions should get you up and running and able to start playing with the large library of examples provided. The impatient will probably have good luck browsing the examples library included with the source, and the *Worked Examples* in this documentation for something that looks like their intended simulation.

If you are upgrading from **XMDS version 1.x**, then after following the installation instructions (*Installation*), you might want to have a quick read of the note for upgraders (*Upgrading From XMDS 1.X*). The syntax of the XML scripts has changed, but hopefully you will find the new scripts very intuitive.

Detailed advice on input/output issues, and ways to code more complicated simulations can be found in *Advanced Topics*.

XMDS2 should be cited as Comput. Phys. Commun. 184, 201-208 (2013).

CHAPTER

TWO

INSTALLATION

XMDS2 can be installed on any unix-like system including Linux, Tru64, and Mac OS X. It requires a C++ compiler, python, and several installed packages. Many of these packages are optional, but a good idea to obtain full functionality.

2.1 Installers

The easiest way to get started is with an installer. If we don't have an installer for your system, follow the *manual installation* instructions.

Linux (Ubuntu/Debian/Fedora/RedHat)	Download Linux Installer	*Learn more*
OS X 10.6/10.7	Download OS X Installer	*Learn more*
Other systems	*Install from source*	

If you have one of the supported operating systems listed above, but you find the installer doesn't work for you, please let us know by emailing xmds-devel <at> lists.sourceforge.net. If you'd like to tweak the linux installer to work on a distribution we haven't tested, we'd love you to do that and let us know!

2.2 Linux installer instructions

The linux installer has currently only been tested with Ubuntu, Debian, Fedora, and Red Hat. Download the installer here: http://xmds.svn.sourceforge.net/viewvc/xmds/trunk/xpdeint/admin/linux_installer.sh

Once you have downloaded it, make the installer executable and run it by typing the following into a terminal:

```
chmod u+x linux_installer.sh
./linux_installer.sh
```

Alternatively, if you wish to download and run the installer in a single step, you can use the following command:

```
/bin/bash -c "$(wget -qO - http://xmds.svn.sf.net/viewvc/xmds/trunk/xpdeint/admin/linux_installer
```

The linux installer installs all XMDS2 dependencies from your native package manager where possible (`apt-get` for Ubuntu/Debian, `yum` for Fedora/Red Hat) but will download and compile the source code for libraries not available through the package manager. This means you'll need to be connected to the internet when running the installer. The installer should not be run with administrative privileges; it will ask you to enter your admin password at the appropriate point.

For instructions on how to install XMDS2 on systems where you lack administrative rights, see *Manual installation from source*.

By default, this installer will install a known stable version of XMDS, which can be updated at any time by navigating to the XMDS directory and typing 'make update'. To install the latest developer version at the beginning, simply run the installer with the `--develop` option.

Once XMDS2 has been installed, you can run it from the terminal by typing `xmds2`. See the *Quickstart Tutorial* for next steps.

2.3 Mac OS X Installation

2.3.1 Download

Mac OS X 10.6 (Snow Leopard) or later XMDS 2 installer: http://sourceforge.net/projects/xmds/files/

2.3.2 Using the Mac OS X Installer

A self-contained installer for Mac OS X 10.6 (Snow Leopard) and later is available from the link above. This installer is only compatible with Intel Macs. This means that the older PowerPC architecture is *not supported*. Xcode (Apple's developer tools) is required to use this installer. Xcode is available for free from the Mac App Store for 10.7 or later, and is available on the install disk of earlier Macs as an optional install. For users of earlier operating systems (10.6.8 or earlier), it is possible to find a free copy of earlier versions of XCode on the Apple developer website (3.2.6 was the Snow Leopard compatible version). You will be prompted to install it if you haven't already.

Once you have downloaded the XMDS installer, installation is as simple as dragging it to your Applications folder or any other location. Click the XMDS application to launch it, and press the "Launch XMDS Terminal" button to open a Terminal window customised to work with XMDS. The first time you do this, the application will complete the installation process. This process can take a few minutes, but is only performed once.

The terminal window launched by the XMDS application has environment variables set for using this installation of XMDS. You can run XMDS in this terminal by typing `xmds2`. See the *Quickstart Tutorial* for next steps.

To uninstall XMDS, drag the XMDS application to the trash. XMDS places some files in the directory `~/Library/XMDS`. Remove this directory to completely remove XMDS from your system.

This package includes binaries for OpenMPI, FFTW, HDF5 and GSL. These binaries are self-contained and do not overwrite any existing installations.

2.4 Manual installation from source

This installation guide will take you through a typical full install step by step. A large part of this procedure is obtaining and installing other libraries that XMDS2 requires, before installing XMDS2 itself.

While the instructions below detail these packages individually, if you have administrative privileges (or can request packages from your administrator) and if you are using an Ubuntu, Debian, Fedora or Red Hat linux distribution, you can install all required and optional dependencies (but not XMDS2 itself) via

Ubuntu / Debian:

```
sudo apt-get install build-essential subversion libopenmpi-dev openmpi-bin python-dev python-setu
```

Fedora / Red Hat:

```
sudo yum install gcc gcc-c++ make automake subversion openmpi-devel python-devel python-setuptool
```

You will still have to download and build FFTW 3.3 from source (see below) since prebuilt packages with MPI and AVX support are not currently available in the repositories.

Also note that this guide adds extra notes for users wishing to install XMDS2 using the SVN repository. This requires a few extra steps, but allows you to edit your copy, and/or update your copy very efficiently (with all the usual advantages and disadvantages of using unreleased material).

0. **You will need a copy of XMDS2.** The current release can be found at Sourceforge, and downloaded as a single file. Download this file, and expand it in a directory where you want to keep the program files.

- Developer-only instructions: You can instead check out a working copy of the source using SVN. In a directory where you want to check out the repository, run: `svn checkout https://xmds.svn.sourceforge.net/svnroot/xmds/trunk/xpdeint .` (Only do this once. To update your copy, type `svn up` or `make update` in the same directory, and then repeat any developer-only instructions below).

1. **You will need a working C++ compiler.** For Mac OS X, this means that the developer tools (XCode) should be installed. One common free compiler is gcc. It can be downloaded using your favourite package manager. XMDS2 can also use Intel's C++ compiler if you have it. Intel's compiler typically generates faster code than gcc, but it isn't free.
2. You will need a python distribution.
   - Mac OS X: It is pre-installed on Mac OS X 10.5 or later.
   - Linux: It should be pre-installed. If not, install using your favourite package manager.

   We require python 2.4 or greater. XMDS2 does not support Python 3.
3. **Install setuptools.** If you have root (sudo) access, the easy way to install this is by executing ez_setup.py from the repository. Simply type `sudo python ez_setup.py`

   If you want to install into your home directory without root access, this is more complex:

   (a) First create the path ~/lib/python2.5/site-packages (assuming you installed python version 2.5) and ~/bin Add "export PYTHONPATH=~/lib/python2.5/site-packages:$PYTHONPATH" and "export PATH=~/bin:$PATH" (if necessary) to your .bashrc file (and run ". ~/.bashrc")

   (b) If necessary install setuptools, by executing ez_setup.py from the repository. `python ez_setup.py --prefix=~`

   If you use Mac OS X 10.5 or later, or installed the Enthought Python Distribution on Windows, then setuptools is already installed. Though if the next step fails, you may need to upgrade setuptools. To do that, type `sudo easy_install -U setuptools`
4. **Install HDF5 and FFTW3 (and optionally MPI).**

   (a) **HDF5 is a library for reading and writing the Hierarchical Data Format.** This is a standardised data format which it is suggested that people use in preference to the older 'binary' output (which is compatible with xmds-1). The advantage of HDF5 is that this data format is understood by a variety of other tools. xsil2graphics2 provides support for loading data created in this format into Mathematica and Matlab.

   XMDS2 only requires the single process version of HDF5, so there is no need to install the MPI version.

   * Sidebar: Installing HDF5 from source follows a common pattern, which you may find yourself repeating later:

   i. After extracting the source directory, type `configure` and then add possible options.

   (For HDF5, install with the `--prefix=/usr/local/` option if you want XMDS2 to find the library automatically. This is rarely needed for other packages.)

   ii. Once that is finished, type `make`. Then wait for that to finish, which will often be longer than you think.

   iii. Finally, type `sudo make install` to install it into the appropriate directory.

   (b) **FFTW is the library XMDS2 uses for Fourier transforms.** This is the transform most people will use in their simulations. If you need support for MPI distributed simulations, you must configure FFTW to use MPI.

   FFTW is available for free at the FFTW website. To configure and compile it, follow the steps described in the HDF5 sidebar above. You may wish to add the `--enable-mpi --disable-fortran` options to the `configure` command.

(c) **MPI is an API for doing parallel processing.** XMDS2 can use MPI to parallelise simulations on multi-processor/multi-core computers, or clusters of computers. Many supercomputing systems come with MPI libraries pre-installed. The Open MPI project has free distributions of this library available.

If you intend to take advantage of XMDS2's multi-processing features, you must install MPI, and configure FFTW3 to use it.

5. There are a range of optional installs. We recommend that you install them all if possible:

   (a) **A Matrix library like ATLAS, Intel's MKL or the GNU Scientific library (GSL)** These libraries allow efficient implementation of transform spaces other than Fourier space. Mac OS X comes with its own (fast) matrix library.

   (b) **numpy is a tool that XMDS2 uses for automated testing.** It can be installed with `sudo easy_install numpy`.

   Mac OS X 10.5 and later come with numpy.

   (c) **lxml is used to validate the syntax of scripts passed to XMDS2.** If you have root access, this can be installed with the command `sudo easy_install lxml`

   You will need to have 'libxml2' and 'libxslt' installed (via your choice of package manager) to install lxml. Sufficient versions are preinstalled on Mac OS X 10.6.

   If you don't have root access or want to install into your home directory, use:
   `easy_install --prefix=~ lxml`

   (d) **h5py is needed for checking the results of XMDS2 tests that generate HDF5 output.** h5py requires numpy version 1.0.3 or later.

   Upgrading h5py on Mac OS X is best done with the source of the package, as the easy_install option can get confused with multiple numpy versions. (Mac OS X Snow Leopard comes with version 1.2.1). After downloading the source, execute `python ./setup.py build` in the source directory, and then `python ./setup.py install` to install it.

6. **Install XMDS2 into your python path by running (in the xmds-2.1.2/ directory):** `sudo ./setup.py develop`

   If you want to install it into your home directory, type `./setup.py develop --prefix=~`

   This step requires access to the net, as it downloads any dependent packages. If you are behind a firewall, you may need to set your HTTP_PROXY environment variable in order to do this.

   - **Developer only instructions:** The Cheetah templates (*.tmpl) must be compiled into python. To do this, run `make` in the xmds-2.1.2/ directory.
   - **Developer-only instructions:** If you have 'numpy' installed, test XMDS2 by typing `./run_tests.py` in the xmds-2.1.2/ directory. The package 'numpy' is one of the optional packages, with installation instructions below.
   - **Developer-only instructions:** To build the user documentation, you first need to install sphinx, either via your package manager or: `sudo easy_install Sphinx`

     Then, to build the documentation, in the xmds-2.1.2/admin/userdoc-source/ directory run: `make html`

     If this results in an error, you may need to run `sudo ./setup.py develop`

     The generated html documentation will then be found at xmds-2.1.2/documentation/index.html

7. Configure XMDS2 by typing `xmds2 --reconfigure`. If XMDS2 is unable to find a library, you can tell XMDS2 where these libraries are located by adding `include` and `lib` search paths using the `--include-path` and `--lib-path` options. For example, if FFTW3 is installed in `/apps/fftw3` with headers in `/apps/fftw3/include/` and the libraries in `/apps/fftw3/lib`, (re)configure XMDS2 by typing:

- `xmds2 --reconfigure --include-path /apps/fftw3/include --lib-path /apps/fftw3/lib`.

If you need to use additional compiler or link flags for XMDS2 to use certain libraries, set the `CXXFLAGS` or `LINKFLAGS` environment variables before calling `xmds2 --reconfigure`. For example, to pass the compiler flag `-pedantic` and the link flag `-lm`, use:

- `CXXFLAGS="-pedantic" LINKFLAGS="-lm" xmds2 --reconfigure`.

Congratulations! You should now have a fully operational copy of xmds2 and xsil2graphics2. You can test your copy using examples from the "xmds-2.1.2/examples" directory, and follow the worked examples in the *Quickstart Tutorial* and *Worked Examples*.

CHAPTER

THREE

QUICKSTART TUTORIAL

In this tutorial, we will create an XMDS2 script to solve the Lorenz Attractor, an example of a dynamical system that exhibits chaos. The equations describing this problem are

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

where we will solve with the parameters $\sigma = 10$, $\rho = 28$, $\beta = \frac{8}{3}$ and the initial condition $x(0) = y(0) = z(0) = 1$.

Below is a script that solves this problem (it's also saved as examples/lorenz.xmds in your XMDS2 directory). Don't worry if it doesn't make sense yet, soon we'll break it down into easily digestible parts.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation xmds-version="2">
  <name>lorenz</name>

  <!-- While not strictly necessary, the following two tags are handy. -->
  <author>Graham Dennis</author>
  <description>
    The Lorenz Attractor, an example of chaos.
  </description>

  <!--
  This element defines some constants.  It can be used for other
  features as well, but we will go into that in more detail later.
  -->
  <features>
    <globals>
        <![CDATA[
        real sigma = 10.0;
        real b = 8.0/3.0;
        real r = 28.0;
        ]]>
     </globals>
   </features>

  <!--
  This part defines all of the dimensions used in the problem,
  in this case, only the dimension of 'time' is needed.
  -->
  <geometry>
    <propagation_dimension> t </propagation_dimension>
  </geometry>
```

```
  <!-- A 'vector' describes the variables that we will be evolving. -->
  <vector name="position" type="real">
    <components>
      x y z
    </components>
    <initialisation>
      <![CDATA[
      x = y = z = 1.0;
      ]]>
    </initialisation>
  </vector>

  <sequence>
    <!--
    Here we define what differential equations need to be solved
    and what algorithm we want to use.
    -->
    <integrate algorithm="ARK89" interval="20.0" tolerance="1e-7">
      <samples>5000</samples>
      <operators>
        <integration_vectors>position</integration_vectors>
        <![CDATA[
        dx_dt = sigma*(y-x);
        dy_dt = r*x - y - x*z;
        dz_dt = x*y - b*z;
        ]]>
      </operators>
    </integrate>
  </sequence>

  <!-- This part defines what data will be saved in the output file -->
  <output format="hdf5" filename="lorenz.xsil">
    <sampling_group initial_sample="yes">
      <moments>xR yR zR</moments>
      <dependencies>position</dependencies>
      <![CDATA[
        xR = x;
        yR = y;
        zR = z;
      ]]>
    </sampling_group>
  </output>
</simulation>
```

You can compile and run this script with **XMDS2**. To compile the script, just pass the name of the script as an argument to **XMDS2**.

```
$ xmds2 lorenz.xmds
xmds2 version 2.1 "Happy Mollusc" (r2680)
Copyright 2000-2012 Graham Dennis, Joseph Hope, Mattias Johnsson
                    and the xmds team
Generating source code...
... done
Compiling simulation...
... done. Type './lorenz' to run.
```

Now we can execute the generated program 'lorenz'.

```
$ ./lorenz
Sampled field (for moment group #1) at t = 0.000000e+00
Sampled field (for moment group #1) at t = 4.000000e-03
Current timestep: 4.000000e-03
Sampled field (for moment group #1) at t = 8.000000e-03
```

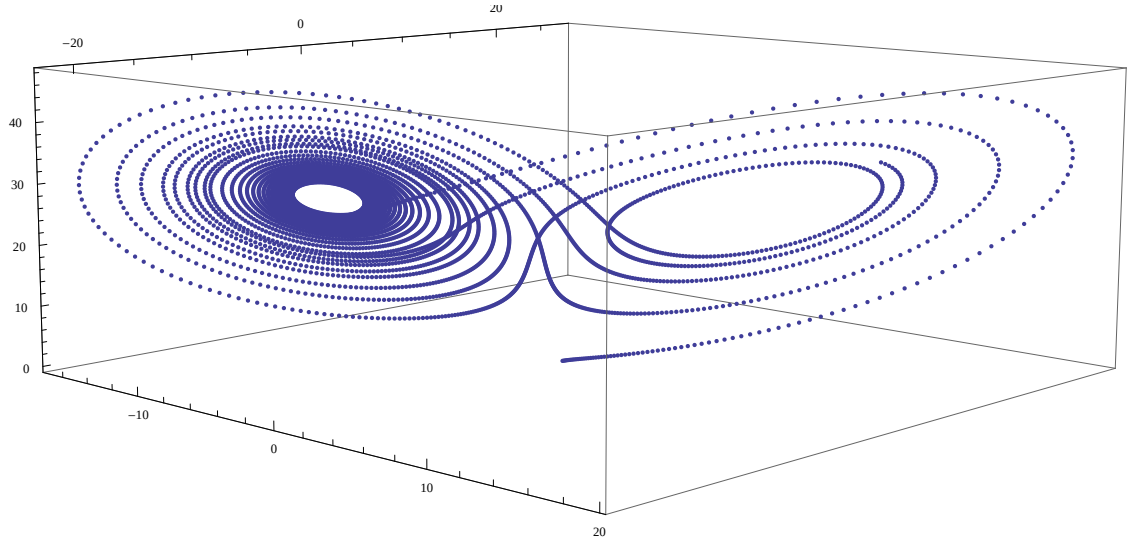
```
Current timestep: 4.000000e-03

... many lines omitted ...

Current timestep: 4.000000e-03
Sampled field (for moment group #1) at t = 1.999600e+01
Current timestep: 4.000000e-03
Sampled field (for moment group #1) at t = 2.000000e+01
Current timestep: 4.000000e-03
Segment 1: minimum timestep: 9.997900e-06 maximum timestep: 4.000000e-03
  Attempted 7386 steps, 0.00% steps failed.
Generating output for lorenz
```

The program generated by **XMDS2** has now integrated your equations and produced two files. The first is the XML file "lorenz.xsil", which contains the all the information used to generate the simulation (including the XMDS2 code) and the metadata description of the output. The second file is named "lorenz.h5", which is a HDF5 file containing all of the output data. You can analysing these files yourself, or import them into your favourite visualisation/postprocessing tool. Here we will use the example of importing it into Mathematica. We run the included utility 'xsil2graphics2'.

```
$ xsil2graphics2 -e lorenz.xsil
xsil2graphics2 from xmds2 version 2.1 "Happy Mollusc" (r2680)
Generating output for Mathematica 6+.
Writing import script for 'lorenz.xsil' to 'lorenz.nb'.
```

This has now generated the file 'lorenz.nb', which is a Mathematica notebook that loads the output data of the simulation. Loading it into Mathematica allows us to plot the points {xR1, yR1, zR1}:

```
ll = Transpose[{xR1, yR1, zR1}];
ListPointPlot3D[ll]
```



...and we see the lobes of the strange attractor. Now let us examine the code that produced this simulation.

First, we have the top level description of the code.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation xmds-version="2">
  <name>lorenz</name>

  <!-- While not strictly necessary, the following two tags are handy. -->
  <author>Graham Dennis</author>
  <description>
    The Lorenz Attractor, an example of chaos.
  </description>
```

One of the advantages of an XML format is that these tags are almost entirely self-explanatory. XMDS2 files follow full XML syntax, so elements can be commented out using the <!-- and --> brackets, and we have an example of that here.

The first line, <?xml ...>, just specifies the encoding and XML version. It is optional, but its presence helps some text editors perform the correct syntax highlighting.

The <simulation> element is mandatory, and encloses the entire simulation script.

The <name> element is optional, but recommended. It defines the name of the executable program that will be generated, as well as the default name of the output data files (although this can be over-ridden in the <output> element if desired). If <name> is not present, it will default to the filename of the script.

The next element we have used can be skipped entirely if you wish to use the default set of features and you don't want to define any global constants for your simulation.

```
<features>
  <globals>
      <![CDATA[
      real sigma = 10.0;
      real b = 8.0/3.0;
      real r = 28.0;
      ]]>
   </globals>
 </features>
```

The <features> element can be used to choose a large number of features that will be discussed later, but here we have only used it to define a <globals> element. This element contains a block of text with <![CDATA[at the start and]]> at the end. These 'CDATA' blocks are used in several places in an XMDS script, and define a block of text that will be pasted directly into the generated C-code. They must therefore be formatted in legal C-syntax, and any legal C-syntax can be used. The <globals> element is placed at the top of the generated code, and can therefore be used to define any variables used in any other part of the simulation. Here we have defined our three real parameters. It is also possible to define variables that can be passed into the program at run-time, an example of which is given in the *Wigner Function* worked example.

The next element is the essential <geometry> element.

```
<geometry>
  <propagation_dimension> t </propagation_dimension>
</geometry>
```

This element is used to define all the dimensions in the problem. We only require the time dimension, which we are labelling 't', so this is a trivial example. We will discuss transverse dimensions in more detail in the next worked example (*The nonlinear Schrödinger equation*), where we deal with the integration of a partial differential equation rather than ordinary differential equations.

Next, we have the <vector> element.

```
<vector name="position" type="real">
  <components>
    x y z
  </components>
  <initialisation>
    <![CDATA[
    x = y = z = 1.0;
    ]]>
  </initialisation>
</vector>
```

We can define multiple vectors, but here we only need the variables that we wish to integrate. We named this vector "position", as it defines the position in phase space. These variables are real-valued (as opposed to, say, complex numbers), so we define type="real". The <components> element defines the names of the elements of this vector, which we have called 'x', 'y' and 'z'. Finally, we provide the initial values of the variables in a CDATA block within the <initialisation> element.

Now we come to the heart of the simulation, where we define the evolution of our vector. This evolution is held in the <sequence> element, which contains an ordered sequence of actions upon any defined vectors. Vectors can be altered with a <filter> element, or integrated in the propagation dimension with an <integrate> element.

```
<sequence>
  <integrate algorithm="ARK89" interval="20.0" tolerance="1e-7">
    <samples>5000</samples>
    <operators>
      <integration_vectors>position</integration_vectors>
      <![CDATA[
      dx_dt = sigma*(y-x);
      dy_dt = r*x - y - x*z;
      dz_dt = x*y - b*z;
      ]]>
    </operators>
  </integrate>
</sequence>
```

Here our sequence consists of a single <integrate> element. It contains several important pieces of information. At the heart, the <operators> element contains the equations of motion as described above, written in a very human-readable fashion. It also contains an <integration_vectors> element, which defines which vectors are used in this integrate block. We have only one vector defined in this simulation, so it is a trivial choice here.

All integrate blocks must define which algorithm is to be used - in this case the 8th (embedded 9th) order adaptive Runge-Kutta method, called "ARK89". The details of different algorithms will be described later (FIXME: Link!), but for now all we need to know is that this algorithm requires a tolerance, and that smaller means more accurate, so we'll make it 10^{-7} by setting `tolerance="1.0e-7"`. Finally, any integration will proceed a certain length in the propagation dimension, which is defined by the "interval" variable. This integrate block will therefore integrate the equations it contains with respect to the propagation dimension ('t') for 20.

The <samples> element says that the values of the output groups will be sampled 5000 times during this interval. The nature of the output is defined in the last element in the simulation: the <output> element.

```
<output format="hdf5" filename="lorenz.xsil">
  <sampling_group initial_sample="yes">
    <moments>xR yR zR</moments>
    <dependencies>position</dependencies>
    <![CDATA[
      xR = x;
      yR = y;
      zR = z;
    ]]>
  </sampling_group>
</output>
```

The two top-level arguments in the <output> element are "format" and "filename". Here we define the output filename, although it would have defaulted to this value. We also choose the format to be HDF5, which is why the simulation resulted in the binary file "lorenz.h5" as well as "lorenz.xsil". If we had instead said `format="ascii"`, then all of the output data would have been written in text form in "lorenz.xsil".

The <output> element can contain any non-zero number of <sampling_group> elements, which specify the entire output of the program. They allow for subsampling, integration of some or all of the transverse dimensions, and/or conversion of some dimensions into Fourier space, but these will be described in more detail in the following examples. We have a <dependencies> element that specifies which vectors are needed for this output. We specify the list of output variables with a <moments> element, and then define them in CDATA block. In this case, we are simply defining the three variables that define our phase space.

And that's it. This is quite a large framework to integrate three coupled ordinary differential equations, but the advantage of using XMDS2 is that vastly more complicated simulations can be performed without increasing the length or complexity of the XMDS2 script significantly. The *Worked Examples* section will provide more complicated examples with stochastic equations and partial differential equations. If you are moved to solve your

own problem using XMDS2, then perhaps the most efficient method will be to take one of the worked examples and adapt it to your needs. All of the examples in the documentation can be found in the "/examples" folder included with the installation.

CHAPTER

FOUR

WORKED EXAMPLES

One of the best ways to learn XMDS2 is to see several illustrative examples. Here are a set of example scripts and explanations of the code, which will be a good way to get started. As an instructional aid, they are meant to be read sequentially, but the adventurous could try starting with one that looked like a simulation they wanted to run, and adapt for their own purposes.

The nonlinear Schrödinger equation (partial differential equation)

Kubo Oscillator (stochastic differential equations)

Fibre Noise (stochastic partial differential equation using parallel processing)

Integer Dimensions (integer dimensions)

Wigner Function (two dimensional PDE using parallel processing, passing arguments in at run time)

Finding the Ground State of a BEC (continuous renormalisation) (PDE with continual renormalisation - computed vectors, filters, breakpoints)

Finding the Ground State of a BEC again (Hermite-Gaussian basis)

Multi-component Schrödinger equation (combined integer and continuous dimensions with matrix multiplication, aliases)

All of these scripts are available in the included "examples" folder, along with more examples that demonstrate other tricks. Together, they provide starting points for a huge range of different simulations.

4.1 The nonlinear Schrödinger equation

This worked example will show a range of new features that can be used in an **XMDS2** script, and we will also examine our first partial differential equation. We will take the one dimensional nonlinear Schrödinger equation, which is a common nonlinear wave equation. The equation describing this problem is:

$$\frac{\partial \phi}{\partial \xi} = \frac{i}{2}\frac{\partial^2 \phi}{\partial \tau^2} - \Gamma(\tau)\phi + i|\phi|^2\phi$$

where ϕ is a complex-valued field, and $\Gamma(\tau)$ is a τ-dependent damping term. Let us look at an XMDS2 script that integrates this equation, and then examine it in detail.

```
<simulation xmds-version="2">
  <name>nlse</name>

  <author>Joe Hope</author>
  <description>
    The nonlinear Schrodinger equation in one dimension,
    which is a simple partial differential equation.
    We introduce several new features in this script.
  </description>
```

```
<features>
    <benchmark />
    <bing />
    <fftw plan="patient" />
    <openmp />
    <auto_vectorise />
    <globals>
        <![CDATA[
        const double energy = 4;
        const double vel = 0.3;
        const double hwhm = 1.0;
        ]]>
     </globals>
   </features>

<geometry>
    <propagation_dimension> xi </propagation_dimension>
    <transverse_dimensions>
      <dimension name="tau" lattice="128"  domain="(-6, 6)" />
    </transverse_dimensions>
 </geometry>

<vector name="wavefunction" type="complex" dimensions="tau">
  <components> phi </components>
  <initialisation>
    <![CDATA[
    const double w0 = hwhm*sqrt(2/log(2));
    const double amp = sqrt(energy/w0/sqrt(M_PI/2));
    phi = amp*exp(-tau*tau/w0/w0)*exp(i*vel*tau);
    ]]>
  </initialisation>
</vector>

<vector name="dampingVector" type="real">
  <components> Gamma </components>
  <initialisation>
    <![CDATA[
    Gamma=1.0*(1-exp(-pow(tau*tau/4.0/4.0,10)));
    ]]>
  </initialisation>
</vector>

<sequence>
  <integrate algorithm="ARK45" interval="20.0" tolerance="1e-7">
    <samples>10 100 10</samples>
    <operators>
      <integration_vectors>wavefunction</integration_vectors>
      <operator kind="ex" constant="yes">
        <operator_names>Ltt</operator_names>
        <![CDATA[
          Ltt = -i*ktau*ktau*0.5;
        ]]>
      </operator>
      <![CDATA[
      dphi_dxi = Ltt[phi] - phi*Gamma + i*mod2(phi)*phi;
      ]]>
      <dependencies>dampingVector</dependencies>
    </operators>
  </integrate>
</sequence>

<output>
```

```
    <sampling_group basis="tau" initial_sample="yes">
      <moments>density</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        density = mod2(phi);
      ]]>
    </sampling_group>

    <sampling_group basis="tau(0)" initial_sample="yes">
      <moments>normalisation</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        normalisation = mod2(phi);
      ]]>
    </sampling_group>

    <sampling_group basis="ktau(32)" initial_sample="yes">
      <moments>densityK</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        densityK = mod2(phi);
      ]]>
    </sampling_group>

  </output>
</simulation>
```

Let us examine the new items in the <features> element that we have demonstrated here. The existence of the <benchmark> element causes the simulation to be timed. The <bing> element causes the computer to make a sound upon the conclusion of the simulation. The <fftw> element is used to pass options to the FFTW libraries for fast Fourier transforms, which are needed to do spectral derivatives for the partial differential equation. Here we used the option *plan="patient"*, which makes the simulation test carefully to find the fastest method for doing the FFTs. More information on possible choices can be found in the FFTW documentation.

Finally, we use two tags to make the simulation run faster. The <auto_vectorise> element switches on several loop optimisations that exist in later versions of the GCC compiler. The <openmp> element turns on threaded parallel processing using the OpenMP standard where possible. These options are not activated by default as they only exist on certain compilers. If your code compiles with them on, then they are recommended.

Let us examine the <geometry> element.

```
<geometry>
    <propagation_dimension> xi </propagation_dimension>
    <transverse_dimensions>
      <dimension name="tau" lattice="128"  domain="(-6, 6)" />
    </transverse_dimensions>
 </geometry>
```

This is the first example that includes a transverse dimension. We have only one dimension, and we have labelled it "tau". It is a continuous dimension, but only defined on a grid containing 128 points (defined with the lattice variable), and on a domain from -6 to 6. The default is that transforms in continuous dimensions are fast Fourier transforms, which means that this dimension is effectively defined on a loop, and the "tau=-6" and "tau=6" positions are in fact the same. Other transforms are possible, as are discrete dimensions such as an integer-valued index, but we will leave these advanced possibilities to later examples.

Two vector elements have been defined in this simulation. One defines the complex-valued wavefunction "phi" that we wish to evolve. We define the transverse dimensions over which this vector is defined by the dimensions tag in the description. By default, it is defined over all of the transverse dimensions in the <geometry> element, so even though we have omitted this tag for the second vector, it also assumes that the vector is defined over all of tau.

The second vector element contains the component "Gamma" which is a function of the transverse variable tau, as specified in the equation of motion for the field. This second vector could have been avoided in two ways. First,

the function could have been written explicitly in the integrate block where it is required, but calculating it once and then recalling it from memory is far more efficient. Second, it could have been included in the "wavefunction" vector as another component, but then it would have been unnecessarily complex-valued, it would have needed an explicit derivative in the equations of motion (presumably `dGamma_dxi = 0;`), and it would have been Fourier transformed whenever the phi component was transformed. So separating it as its own vector is far more efficient.

The `<integrate>` element for a partial differential equation has some new features:

```
<integrate algorithm="ARK45" interval="20.0" tolerance="1e-7">
  <samples>10 100 10</samples>
  <operators>
    <integration_vectors>wavefunction</integration_vectors>
    <operator kind="ex" constant="yes">
      <operator_names>Ltt</operator_names>
      <![CDATA[
        Ltt = -i*ktau*ktau*0.5;
      ]]>
    </operator>
    <![CDATA[
    dphi_dxi = Ltt[phi] - phi*Gamma + i*mod2(phi)*phi;
    ]]>
    <dependencies>dampingVector</dependencies>
  </operators>
</integrate>
```

There are some trivial changes from the tutorial script, such as the fact that we are using the ARK45 algorithm rather than ARK89. Higher order algorithms are often better, but not always. Also, since this script has multiple output groups, we have to specify how many times each of these output groups are sampled in the `<samples>` element, so there are three numbers there. Besides the vectors that are to be integrated, we also specify that we want to use the vector "dampingVector" during this integration. This is achieved by including the `<dependencies>` element inside the `<operators>` element.

The equation of motion as written in the CDATA block looks almost identical to our desired equation of motion, except for the term based on the second derivative, which introduces an important new concept. Inside the `<operators>` element, we can define any number of operators. Operators are used to define functions in the transformed space of each dimension, which in this case is Fourier space. The derivative of a function is equivalent to multiplying by $-i * k$ in Fourier space, so the $\frac{i}{2}\frac{\partial^2 \phi}{\partial \tau^2}$ term in our equation of motion is equivalent to multiplying by $-\frac{i}{2}k_\tau^2$ in Fourier space. In this example we define "Ltt" as an operator of exactly that form, and in the equation of motion it is applied to the field "phi".

Operators can be explicit (`kind="ex"`) or in the interaction picture (`kind="ip"`). The interaction picture can be more efficient, but it restricts the possible syntax of the equation of motion. Safe utilisation of interaction picture operators will be described later, but for now let us emphasise that **explicit operators should be used** unless the user is clear what they are doing. That said, **XMDS2** will generate an error if the user tries to use interaction picture operators incorrectly. The `constant="yes"` option in the operator block means that the operator is not a function of the propagation dimension "xi", and therefore only needs to be calculated once at the start of the simulation.

The output of a partial differential equation offers more possibilities than an ordinary differential equation, and we examine some in this example.

For vectors with transverse dimensions, we can sample functions of the vectors on the full lattice or a subset of the points. In the `<sampling_group>` element, we must add a string called "basis" that determines the space in which each transverse dimension is to be sampled, optionally followed by the number of points to be sampled in parentheses. If the number of points is not specified, it will default to a complete sampling of all points in that dimension. If a non-zero number of points is specified, it must be a factor of the lattice size for that dimension.

```
<sampling_group basis="tau" initial_sample="yes">
  <moments>density</moments>
  <dependencies>wavefunction</dependencies>
  <![CDATA[
    density = mod2(phi);
```

```
  ]]>
</sampling_group>
```

The first output group samples the mod square of the vector "phi" over the full lattice of 128 points.

If the lattice parameter is set to zero points, then the corresponding dimension is integrated.

```
<sampling_group basis="tau(0)" initial_sample="yes">
  <moments>normalisation</moments>
  <dependencies>wavefunction</dependencies>
  <![CDATA[
    normalisation = mod2(phi);
  ]]>
</sampling_group>
```

This second output group samples the normalisation of the wavefunction $\int d\tau |\phi(\tau)|^2$ over the domain of τ. This output requires only a single real number per sample, so in the integrate element we have chosen to sample it many more times than the vectors themselves.

Finally, functions of the vectors can be sampled with their dimensions in Fourier space.

```
<sampling_group basis="ktau(32)" initial_sample="yes">
  <moments>densityK</moments>
  <dependencies>wavefunction</dependencies>
  <![CDATA[
    densityK = mod2(phi);
  ]]>
</sampling_group>
```

The final output group above samples the mod square of the Fourier-space wavefunction phi on a sample of 32 points.

4.2 Kubo Oscillator

This example demonstrates the integration of a stochastic differential equation. We examine the Kubo oscillator, which is a complex variable whose phase is evolving according to a Wiener noise. In a suitable rotating frame, the equation of motion for the variable is

$$\frac{dz}{dt} = iz\,\eta$$

where $\eta(t)$ is the Wiener differential, and we interpret this as a Stratonovich equation. In other common notation, this is sometimes written:

$$dz = iz \circ dW$$

Most algorithms employed by XMDS require the equations to be input in the Stratonovich form. Ito differential equations can always be transformed into Stratonovich euqations, and in this case the difference is equivalent to the choice of rotating frame. This equation is solved by the following XMDS2 script:

```
<simulation xmds-version="2">
  <name>kubo</name>
  <author>Graham Dennis and Joe Hope</author>
  <description>
    Example Kubo oscillator simulation
  </description>

  <geometry>
    <propagation_dimension> t </propagation_dimension>
```

```
  </geometry>

  <driver name="multi-path" paths="10000" />

  <features>
    <error_check />
    <benchmark />
  </features>

  <noise_vector name="drivingNoise" dimensions="" kind="wiener" type="real" method="dsfmt" seed="
    <components>eta</components>
  </noise_vector>

  <vector name="main" type="complex">
    <components> z </components>
    <initialisation>
      <![CDATA[
        z = 1.0;
      ]]>
    </initialisation>
  </vector>

  <sequence>
    <integrate algorithm="SI" interval="10" steps="1000">
      <samples>100</samples>
      <operators>
        <integration_vectors>main</integration_vectors>
        <dependencies>drivingNoise</dependencies>
        <![CDATA[
          dz_dt = i*z*eta;
        ]]>
      </operators>
    </integrate>
  </sequence>

  <output>
    <sampling_group initial_sample="yes">
      <moments>zR zI</moments>
      <dependencies>main</dependencies>
      <![CDATA[
        zR = z.Re();
        zI = z.Im();
      ]]>
    </sampling_group>
  </output>
</simulation>
```

The first new item in this script is the `<driver>` element. This element enables us to change top level management of the simulation. Without this element, XMDS2 will integrate the stochastic equation as described. With this element and the option `name="multi-path"`, it will integrate it multiple times, using different random numbers each time. The output will then contain the mean values and standard errors of your output variables. The number of integrations included in the averages is set with the `paths` variable.

In the `<features>` element we have included the `<error_check>` element. This performs the integration first with the specified number of steps (or with the specified tolerance), and then with twice the number of steps (or equivalently reduced tolerance). The output then includes the difference between the output variables on the coarse and the fine grids as the 'error' in the output variables. This error is particularly useful for stochastic integrations, where algorithms with adaptive step-sizes are less safe, so the number of integration steps must be user-specified.

We define the stochastic elements in a simulation with the `<noise_vector>` element.

```
<noise_vector name="drivingNoise" dimensions="" kind="wiener" type="real" method="dsfmt" seed="31
 <components>eta</components>
</noise_vector>
```

This defines a vector that is used like any other, but it will be randomly generated with particular statistics and characteristics rather than initialised. The name, dimensions and type tags are defined just as for normal vectors. The names of the components are also defined in the same way. The noise is defined as a Wiener noise here (`kind = "wiener"`), which is a zero-mean Gaussian random noise with an average variance equal to the discretisation volume (here it is just the step size in the propagation dimension, as it is not defined over transverse dimensions). Other noise types are possible, including uniform and Poissonian noises, but we will not describe them in detail here.

We may also define a noise method to choose a non-default pseudo random number generator, and a seed for the random number generator. Using a seed can be very useful when debugging the behaviour of a simulation, and many compilers have pseudo-random number generators that are superior to the default option (posix).

The integrate block is using the semi-implicit algorithm (`algorithm="SI"`), which is a good default choice for stochastic problems, even though it is only second order convergent for deterministic equations. More will be said about algorithm choice later, but for now we should note that adaptive algorithms based on Runge-Kutta methods are not guaranteed to converge safely for stochastic equations. This can be particularly deceptive as they often succeed, particularly for almost any problem for which there is a known analytic solution.

We include elements from the noise vector in the equation of motion just as we do for any other vector. The default SI and Runge-Kutta algorithms converge to the *Stratonovich* integral. Ito stochastic equations can be converted to Stratonovich form and vice versa.

Executing the generated program 'kubo' gives slightly different output due to the "multi-path" driver.

```
$ ./kubo
Beginning full step integration ...
Starting path 1
Starting path 2

... many lines omitted ...

Starting path 9999
Starting path 10000
Beginning half step integration ...
Starting path 1
Starting path 2

... many lines omitted ...

Starting path 9999
Starting path 10000
Generating output for kubo
Maximum step error in moment group 1 was 4.942549e-04
Time elapsed for simulation is: 2.71 seconds
```

The maximum step error in each moment group is given in absolute terms. This is the largest difference between the full step integration and the half step integration. While a single path might be very stochastic:

The average over multiple paths can be increasingly smooth.

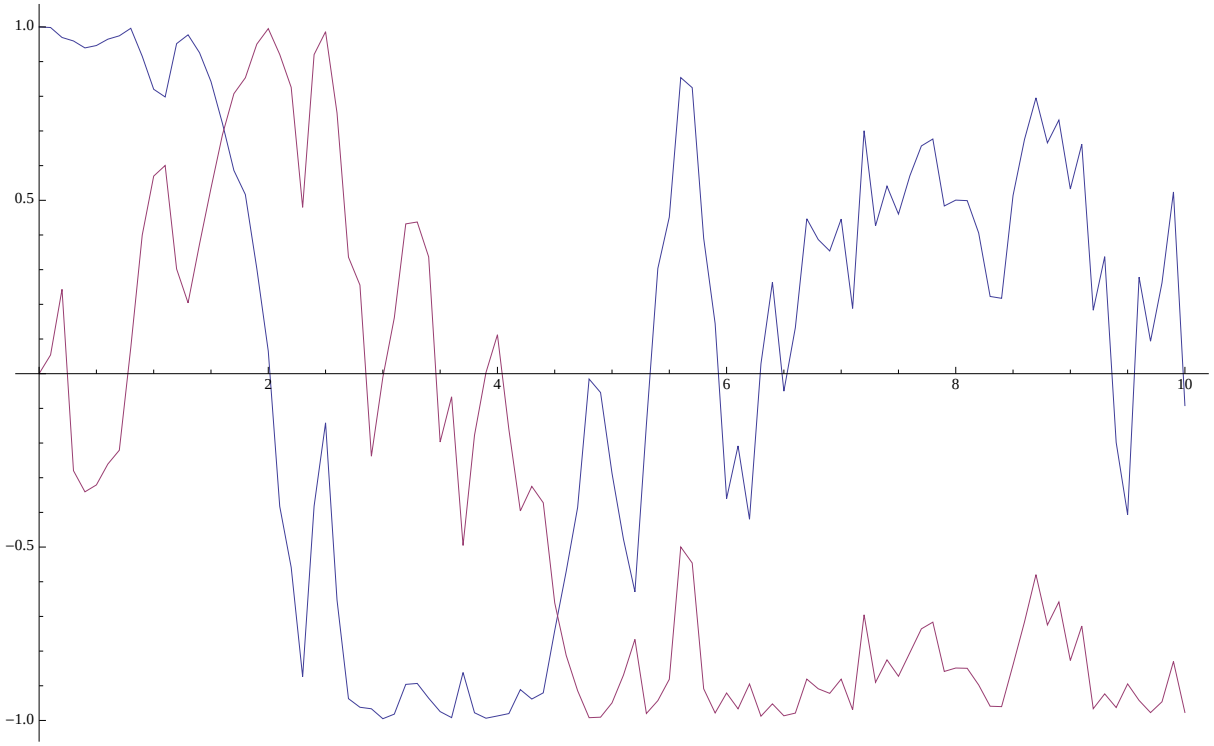


Figure 4.1: The mean value of the real and imaginary components of the z variable for a single path of the simulation.

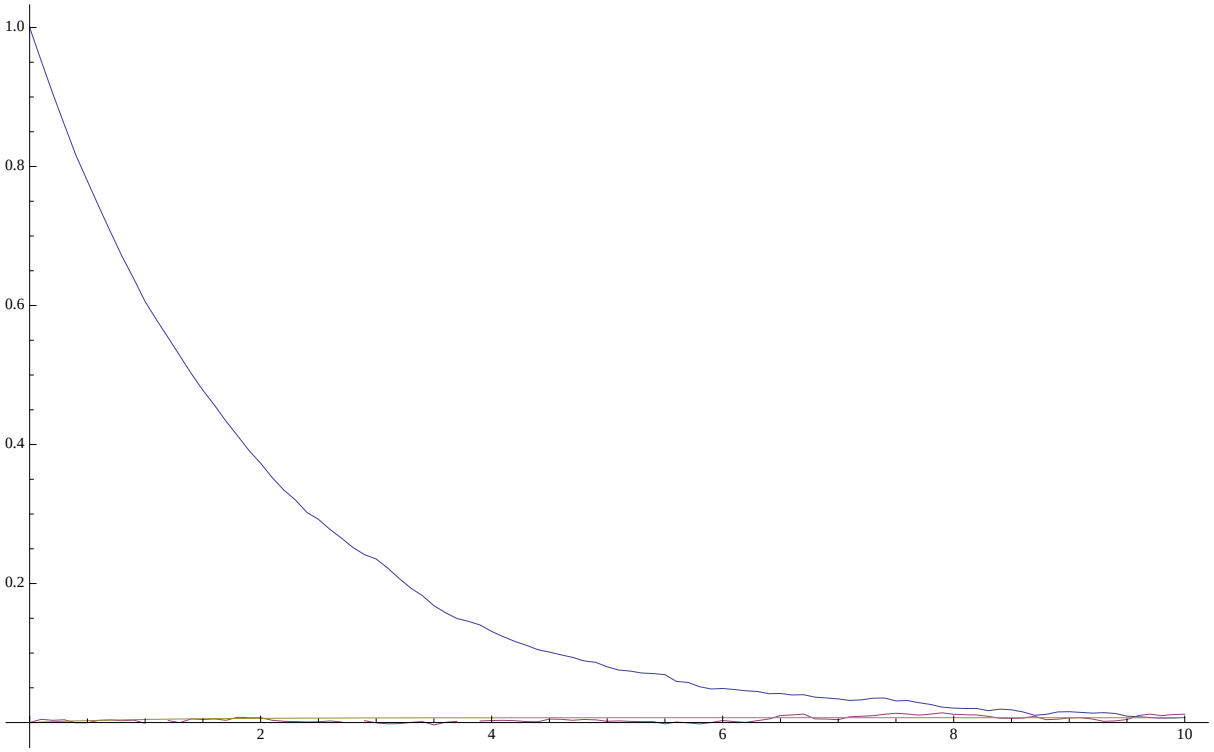


Figure 4.2: The mean and standard error of the z variable averaged over 10000 paths, as given by this simulation. It agrees within the standard error with the expected result of $\exp(-t/2)$.

4.3 Fibre Noise

This simulation is a stochastic partial differential equation, in which a one-dimensional damped field is subject to a complex noise. This script can be found in `examples/fibre.xmds`.

$$\frac{\partial \psi}{\partial t} = -i\frac{\partial^2 \psi}{\partial x^2} - \gamma\psi + \beta\frac{1}{\sqrt{2}}\left(\eta_1(x) + i\eta_2(x)\right)$$

where the noise terms $\eta_j(x,t)$ are Wiener differentials and the equation is interpreted as a Stratonovich differential equation. On a finite grid, these increments have variance $\frac{1}{\Delta x \Delta t}$.

```
<simulation xmds-version="2">
  <name>fibre</name>
  <author>Joe Hope and Graham Dennis</author>
  <description>
    Example fibre noise simulation
  </description>

  <geometry>
    <propagation_dimension> t </propagation_dimension>
    <transverse_dimensions>
      <dimension name="x" lattice="64"  domain="(-5, 5)" />
    </transverse_dimensions>
  </geometry>

  <driver name="mpi-multi-path" paths="8" />

  <features>
    <auto_vectorise />
    <benchmark />
    <error_check />
    <globals>
      <![CDATA[
      const real ggamma = 1.0;
      const real beta = sqrt(M_PI*ggamma/10.0);
      ]]>
    </globals>
  </features>

  <noise_vector name="drivingNoise" dimensions="x" kind="wiener" type="complex" method="dsfmt" se
    <components>Eta</components>
  </noise_vector>

  <vector name="main" initial_basis="x" type="complex">
    <components>phi</components>
    <initialisation>
      <![CDATA[
        phi = 0.0;
      ]]>
    </initialisation>
  </vector>

  <sequence>
    <integrate algorithm="SI" iterations="3" interval="2.5" steps="200000">
      <samples>50</samples>
      <operators>
        <operator kind="ex" constant="yes">
          <operator_names>L</operator_names>
          <![CDATA[
            L = -i*kx*kx;
          ]]>
```

```
        </operator>
        <dependencies>drivingNoise</dependencies>
        <integration_vectors>main</integration_vectors>
        <![CDATA[
          dphi_dt = L[phi] - ggamma*phi + beta*Eta;
        ]]>
      </operators>
    </integrate>
  </sequence>

  <output>
    <sampling_group basis="kx" initial_sample="yes">
      <moments>pow_dens</moments>
      <dependencies>main</dependencies>
      <![CDATA[
        pow_dens = mod2(phi);
      ]]>
    </sampling_group>
  </output>
</simulation>
```

Note that the noise vector used in this example is complex-valued, and has the argument `dimensions="x"` to define it as a field of delta-correlated noises along the x-dimension.

This simulation demonstrates the ease with which XMDS2 can be used in a parallel processing environment. Instead of using the stochastic driver "multi-path", we simply replace it with "mpi-multi-path". This instructs XMDS2 to write a parallel version of the program based on the widespread MPI standard. This protocol allows multiple processors or clusters of computers to work simultaneously on the same problem. Free open source libraries implementing this standard can be installed on a linux machine, and come standard on Mac OS X. They are also common on many supercomputer architectures. Parallel processing can also be used with deterministic problems to great effect, as discussed in the later example *Wigner Function*.

Executing this program is slightly different with the MPI option. The details can change between MPI implementations, but as an example:

```
$xmds2 fibre.xmds
xmds2 version 2.1 "Happy Mollusc" (r2543)
Copyright 2000-2012 Graham Dennis, Joseph Hope, Mattias Johnsson
                    and the xmds team
Generating source code...
... done
Compiling simulation...
... done. Type './fibre' to run.
```

Note that different compile options (and potentially a different compiler) are used by XMDS2, but this is transparent to the user. MPI simulations will have to be run using syntax that will depend on the MPI implementation. Here we show the version based on the popular open source Open-MPI implementation.

```
$ mpirun -np 4 ./fibre
Found enlightenment... (Importing wisdom)
Planning for x <---> kx transform... done.
Beginning full step integration ...
Rank[0]: Starting path 1
Rank[1]: Starting path 2
Rank[2]: Starting path 3
Rank[3]: Starting path 4
Rank[3]: Starting path 8
Rank[0]: Starting path 5
Rank[1]: Starting path 6
Rank[2]: Starting path 7
Rank[3]: Starting path 4
Beginning half step integration ...
Rank[0]: Starting path 1
```

```
Rank[2]: Starting path 3
Rank[1]: Starting path 2
Rank[3]: Starting path 8
Rank[0]: Starting path 5
Rank[2]: Starting path 7
Rank[1]: Starting path 6
Generating output for fibre
Maximum step error in moment group 1 was 4.893437e-04
Time elapsed for simulation is: 20.99 seconds
```

In this example we used four processors. The different processors are labelled by their "Rank", starting at zero. Because the processors are working independently, the output from the different processors can come in a randomised order. In the end, however, the .xsil and data files are constructed identically to the single processor outputs.

The analytic solution to the stochastic averages of this equation is given by

$$\langle|\psi(k,t)|^2\rangle = \exp(-2\gamma t)|\psi(k,0)|^2 + \frac{\beta^2 L_x}{4\pi\gamma}\left(1-\exp(-2\gamma t)\right)$$

where L_x is the length of the x domain. We see that a single integration of these equations is quite chaotic:

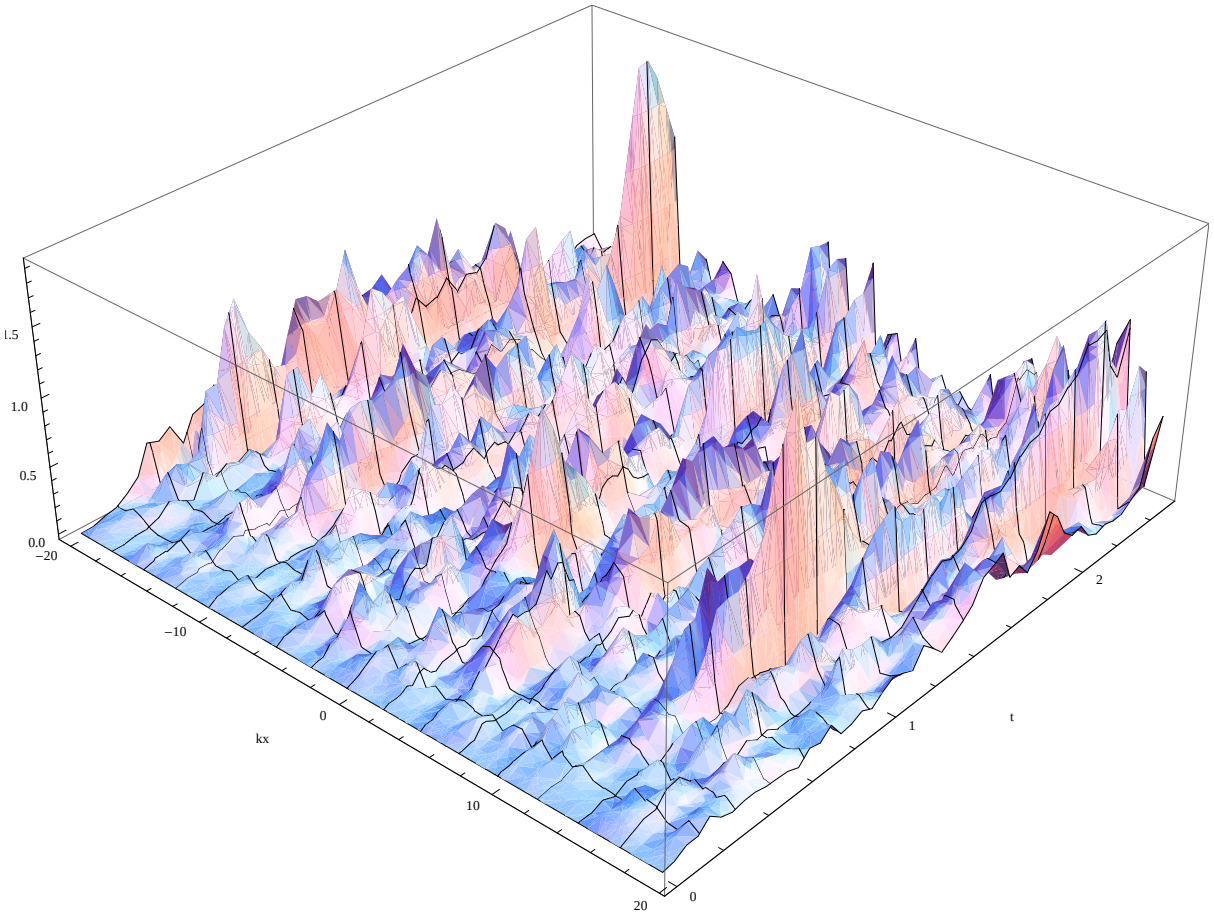


Figure 4.3: The momentum space density of the field as a function of time for a single path realisation.

while an average of 1024 paths (change `paths="8"` to `paths="1024"` in the `<driver>` element) converges nicely to the analytic solution:

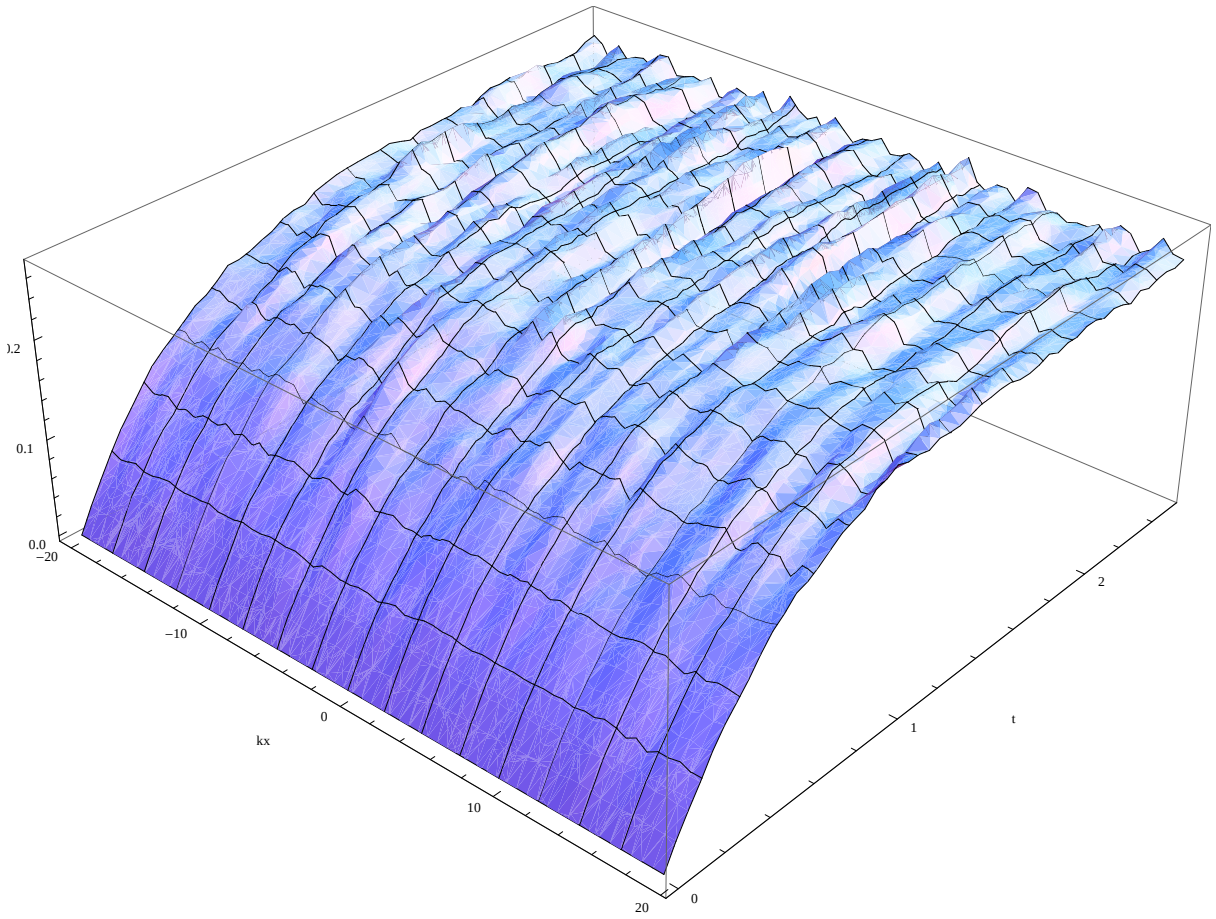


Figure 4.4: The momentum space density of the field as a function of time for an average of 1024 paths.

4.4 Integer Dimensions

This example shows how to handle systems with integer-valued transverse dimensions. We will integrate the following set of equations

$$\frac{dx_j}{dt} = x_j \left(x_{j-1} - x_{j+1}\right)$$

where x_j are complex-valued variables defined on a ring, such that $j \in \{0, j_{max}\}$ and the $x_{j_{max}+1}$ variable is identified with the variable x_0, and the variable x_{-1} is identified with the variable $x_{j_{max}}$.

```
<simulation xmds-version="2">
  <name>integer_dimensions</name>
  <author>Graham Dennis</author>
  <description>
    XMDS2 script to test integer dimensions.
  </description>

  <features>
    <benchmark />
    <error_check />
    <bing />
    <diagnostics /> <!-- This will make sure that all nonlocal accesses of dimensions are safe --
  </features>

  <geometry>
    <propagation_dimension> t </propagation_dimension>
    <transverse_dimensions>
      <dimension name="j" type="integer" lattice="5" domain="(0,4)" />
    </transverse_dimensions>
  </geometry>

  <vector name="main" type="complex">
    <components> x </components>
    <initialisation>
      <![CDATA[
      x = 1.0e-3;
      x(j => 0) = 1.0;
      ]]>
    </initialisation>
  </vector>

  <sequence>
    <integrate algorithm="ARK45" interval="60" steps="25000" tolerance="1.0e-9">
      <samples>1000</samples>
      <operators>
        <integration_vectors>main</integration_vectors>
        <![CDATA[
        long j_minus_one = (j-1) % _lattice_j;
        if (j_minus_one < 0)
          j_minus_one += _lattice_j;
        long j_plus_one  = (j+1) % _lattice_j;
        dx_dt(j => j) = x(j => j)*(x(j => j_minus_one) - x(j => j_plus_one));
        ]]>
      </operators>
    </integrate>
  </sequence>

  <output>
    <sampling_group basis="j" initial_sample="yes">
      <moments>xR</moments>
      <dependencies>main</dependencies>
```

```
      <![CDATA[
        xR = x.Re();
      ]]>
    </sampling_group>
  </output>
</simulation>
```

The first extra feature we have used in this script is the `<diagnostics>` element. It performs run-time checking that our generated code does not accidentally attempt to access a part of our vector that does not exist. Removing this tag will increase the speed of the simulation, but its presence helps catch coding errors.

The simulation defines a vector with a single transverse dimension labelled "j", of type "integer" ("int" and "long" can also be used as synonyms for "integer"). In the absence of an explicit type, the dimension is assumed to be real-valued. The dimension has a "domain" argument as normal, defining the minimum and maximum values of the dimension's range. The lattice element, if specified, is used as a check on the size of the domain, and will create an error if the two do not match.

Integer-valued dimensions can be called non-locally. Real-valued dimensions are typically coupled non-locally only through local operations in the transformed space of the dimension, but can be called non-locally in certain other situations as described in *the reference*. The syntax for calling integer dimensions non-locally can be seen in the initialisation CDATA block:

```
x = 1.0e-3;
x(j => 0) = 1.0;
```

where the syntax `x(j => 0)` is used to reference the variable x_0 directly. We see a more elaborate example in the integrate CDATA block:

```
dx_dt(j => j) = x(j => j)*(x(j => j_minus_one) - x(j => j_plus_one));
```

where the vector "x" is called using locally defined variables. This syntax is chosen so that multiple dimensions can be addressed non-locally with minimal possibility for confusion.

4.5 Wigner Function

This example integrates the two-dimensional partial differential equation

$$\frac{\partial W}{\partial t} = \left[\left(\omega + \frac{U_{int}}{\hbar} \left(x^2 + y^2 - 1 \right) \right) \left(x \frac{\partial}{\partial y} - y \frac{\partial}{\partial x} \right) - \frac{U_{int}}{16\hbar} \left(x \left(\frac{\partial^3}{\partial x^2 \partial y} + \frac{\partial^3}{\partial y^3} \right) - y \left(\frac{\partial^3}{\partial y^2 \partial x} + \frac{\partial^3}{\partial x^3} \right) \right) \right] W(x, y, t)$$

with the added restriction that the derivative is forced to zero outside a certain radius. This extra condition helps maintain the long-term stability of the integration. The script can be found in `examples/wigner_arguments_mpi.xmds` under your XMDS2 installation directory.

```
<simulation xmds-version="2">
  <name>wigner</name>
  <author>Graham Dennis and Joe Hope</author>
  <description>
    Simulation of the Wigner function for an anharmonic oscillator with the initial state
    being a coherent state.
  </description>
  <features>
    <benchmark />
    <globals>
      <![CDATA[
        real Uint_hbar_on16;
```

```
    ]]>
  </globals>
  <arguments>
    <argument name="omega" type="real" default_value="0.0" />
    <argument name="alpha_0"     type="real" default_value="3.0" />
    <argument name="absorb"     type="real" default_value="8.0" />
    <argument name="width" type="real" default_value="0.3" />
    <argument name="Uint_hbar" type="real" default_value="1.0" />
    <![CDATA[
      /* derived constants */
      Uint_hbar_on16 = Uint_hbar/16.0;
    ]]>
  </arguments>
  <bing />
  <fftw plan="patient" />
  <openmp />
</features>

<driver name="distributed-mpi" />

<geometry>
  <propagation_dimension> t </propagation_dimension>
  <transverse_dimensions>
    <dimension name="x" lattice="128"  domain="(-6, 6)" />
    <dimension name="y" lattice="128"  domain="(-6, 6)" />
  </transverse_dimensions>
</geometry>

<vector name="main" initial_basis="x y" type="complex">
  <components> W </components>
  <initialisation>
    <![CDATA[
      W = 2.0/M_PI * exp(-2.0*(y*y + (x-alpha_0)*(x-alpha_0)));
    ]]>
  </initialisation>
</vector>

<vector name="dampConstants" initial_basis="x y" type="real">
  <components>damping</components>
  <initialisation>
    <![CDATA[
    if (sqrt(x*x + y*y) > _max_x-width)
      damping = 0.0;
    else
      damping = 1.0;
    ]]>
  </initialisation>
</vector>

<sequence>
  <integrate algorithm="ARK89" tolerance="1e-7" interval="7.0e-4" steps="100000">
    <samples>50</samples>
    <operators>
      <operator kind="ex" constant="yes">
        <operator_names>Lx Ly Lxxx Lxxy Lxyy Lyyy</operator_names>
        <![CDATA[
          Lx = i*kx;
          Ly = i*ky;
          Lxxx = -i*kx*kx*kx;
          Lxxy = -i*kx*kx*ky;
          Lxyy = -i*kx*ky*ky;
          Lyyy = -i*ky*ky*ky;
        ]]>
```

```
        </operator>
        <integration_vectors>main</integration_vectors>
        <dependencies>dampConstants</dependencies>
        <![CDATA[
        real rotation = omega + Uint_hbar*(-1.0 + x*x + y*y);

        dW_dt = damping * ( rotation * (x*Ly[W] - y*Lx[W])
                    - Uint_hbar_on16*( x*(Lxxy[W] + Lyyy[W]) - y*(Lxyy[W] + Lxxx[W]) )
                );
        ]]>
      </operators>
    </integrate>
  </sequence>

  <output>
    <sampling_group basis="x y" initial_sample="yes">
      <moments>WR WI</moments>
      <dependencies>main</dependencies>
      <![CDATA[
        _SAMPLE_COMPLEX(W);
      ]]>
    </sampling_group>
  </output>
</simulation>
```

This example demonstrates two new features of XMDS2. The first is the use of parallel processing for a deterministic problem. The FFTW library only allows MPI processing of multidimensional vectors. For multidimensional simulations, the generated program can be parallelised simply by adding the `name="distributed-mpi"` argument to the `<driver>` element.

```
$ xmds2 wigner_argument_mpi.xmds
xmds2 version 2.1 "Happy Mollusc" (r2680)
Copyright 2000-2012 Graham Dennis, Joseph Hope, Mattias Johnsson
                    and the xmds team
Generating source code...
... done
Compiling simulation...
... done. Type './wigner' to run.
```

To use multiple processors, the final program is then called using the (implementation specific) MPI wrapper:

```
$ mpirun -np 2 ./wigner
Planning for (distributed x, y) <---> (distributed ky, kx) transform... done.
Planning for (distributed x, y) <---> (distributed ky, kx) transform... done.
Sampled field (for moment group #1) at t = 0.000000e+00
Current timestep: 5.908361e-06
Sampled field (for moment group #1) at t = 1.400000e-05
Current timestep: 4.543131e-06

...
```

The possible acceleration achievable when parallelising a given simulation depends on a great many things including available memory and cache. As a general rule, it will improve as the simulation size gets larger, but the easiest way to find out is to test. The optimum speed up is obviously proportional to the number of available processing cores.

The second new feature in this simulation is the `<arguments>` element in the `<features>` block. This is a way of specifying global variables with a given type that can then be input at run time. The variables are specified in a self explanatory way

```
<arguments>
  <argument name="omega" type="real" default_value="0.0" />
    ...
```

```
  <argument name="Uint_hbar" type="real" default_value="1.0" />
</arguments>
```

where the "default_value" is used as the valuable of the variable if no arguments are given. In the absence of the generating script, the program can document its options with the `--help` argument:

```
$ ./wigner --help
Usage: wigner --omega <real> --alpha_0 <real> --absorb <real> --width <real> --Uint_hbar <real>

Details:
Option              Type            Default value
-o,  --omega        real            0.0
-a,  --alpha_0      real            3.0
-b,  --absorb       real            8.0
-w,  --width        real            0.3
-U,  --Uint_hbar    real            1.0
```

We can change one or more of these variables' values in the simulation by passing it at run time.

```
$ mpirun -np 2 ./wigner --omega 0.1 --alpha_0 2.5 --Uint_hbar 0
Found enlightenment... (Importing wisdom)
Planning for (distributed x, y) <---> (distributed ky, kx) transform... done.
Planning for (distributed x, y) <---> (distributed ky, kx) transform... done.
Sampled field (for moment group #1) at t = 0.000000e+00
Current timestep: 1.916945e-04

...
```

The values that were used for the variables, whether default or passed in, are stored in the output file (wigner.xsil).

```
<info>
Script compiled with XMDS2 version 2.1 "Happy Mollusc" (r2680)
See http://www.xmds.org for more information.

Variables that can be specified on the command line:
  Command line argument omega = 1.000000e-01
  Command line argument alpha_0 = 2.500000e+00
  Command line argument absorb = 8.000000e+00
  Command line argument width = 3.000000e-01
  Command line argument Uint_hbar = 0.000000e+00
</info>
```

Finally, note the shorthand used in the output group

```
<![CDATA[
  _SAMPLE_COMPLEX(W);
]]>
```

which is short for

```
<![CDATA[
  WR = W.Re();
  WI = W.Im();
]]>
```

4.6 Finding the Ground State of a BEC (continuous renormalisation)

This simulation solves another partial differential equation, but introduces several powerful new features in XMDS2. The nominal problem is the calculation of the lowest energy eigenstate of a non-linear Schrödinger

equation:

$$\frac{\partial \phi}{\partial t} = i\left[\frac{1}{2}\frac{\partial^2}{\partial y^2} - V(y) - U_{int}|\phi|^2\right]\phi$$

which can be found by evolving the above equation in imaginary time while keeping the normalisation constant. This causes eigenstates to exponentially decay at the rate of their eigenvalue, so after a short time only the state with the lowest eigenvalue remains. The evolution equation is straightforward:

$$\frac{\partial \phi}{\partial t} = \left[\frac{1}{2}\frac{\partial^2}{\partial y^2} - V(y) - U_{int}|\phi|^2\right]\phi$$

but we will need to use new XMDS2 features to manage the normalisation of the function $\phi(y,t)$. The normalisation for a non-linear Schrödinger equation is given by $\int dy|\phi(y,t)|^2 = N_{particles}$, where $N_{particles}$ is the number of particles described by the wavefunction.

The code for this simulation can be found in `examples/groundstate_workedexamples.xmds`:

```
<simulation xmds-version="2">
  <name>groundstate</name>
  <author>Joe Hope</author>
  <description>
    Calculate the ground state of the non-linear Schrodinger equation in a harmonic magnetic trap
    This is done by evolving it in imaginary time while re-normalising each timestep.
  </description>

  <features>
    <auto_vectorise />
    <benchmark />
    <bing />
    <fftw plan="exhaustive" />
    <globals>
      <![CDATA[
        const real Uint = 2.0;
        const real Nparticles = 5.0;
      ]]>
    </globals>
  </features>

  <geometry>
    <propagation_dimension> t </propagation_dimension>
    <transverse_dimensions>
      <dimension name="y" lattice="256"  domain="(-15.0, 15.0)" />
    </transverse_dimensions>
  </geometry>

  <vector name="potential" initial_basis="y" type="real">
    <components> V1 </components>
    <initialisation>
      <![CDATA[
        V1  = 0.5*y*y;
      ]]>
    </initialisation>
  </vector>

  <vector name="wavefunction" initial_basis="y" type="complex">
    <components> phi </components>
    <initialisation>
      <![CDATA[
        if (fabs(y) < 3.0) {
          phi = 1.0;
```

```
        // This will be automatically normalised later
      } else {
        phi = 0.0;
      }
        ]]>
  </initialisation>
</vector>

<computed_vector name="normalisation" dimensions="" type="real">
  <components> Ncalc </components>
  <evaluation>
    <dependencies basis="y">wavefunction</dependencies>
    <![CDATA[
      // Calculate the current normalisation of the wave function.
      Ncalc = mod2(phi);
    ]]>
  </evaluation>
</computed_vector>

<sequence>
    <filter>
      <![CDATA[
        printf("Hello world from a filter segment!\n");
      ]]>
    </filter>

  <filter>
      <dependencies>normalisation wavefunction</dependencies>
    <![CDATA[
      phi *= sqrt(Nparticles/Ncalc);
    ]]>
  </filter>

  <integrate algorithm="ARK45" interval="1.0" steps="4000" tolerance="1e-10">
    <samples>25 4000</samples>
    <filters where="step end">
      <filter>
        <dependencies>wavefunction normalisation</dependencies>
        <![CDATA[
          // Correct normalisation of the wavefunction
          phi *= sqrt(Nparticles/Ncalc);
        ]]>
      </filter>
    </filters>
    <operators>
      <operator kind="ip" constant="yes">
        <operator_names>T</operator_names>
        <![CDATA[
          T = -0.5*ky*ky;
        ]]>
      </operator>
      <integration_vectors>wavefunction</integration_vectors>
      <dependencies>potential</dependencies>
      <![CDATA[
        dphi_dt = T[phi] - (V1 + Uint*mod2(phi))*phi;
      ]]>
    </operators>
  </integrate>

  <breakpoint filename="groundstate_break.xsil">
    <dependencies basis="ky">wavefunction </dependencies>
  </breakpoint>
```

```
  </sequence>

  <output>
    <sampling_group basis="y" initial_sample="yes">
      <moments>norm_dens</moments>
      <dependencies>wavefunction normalisation</dependencies>
      <![CDATA[
        norm_dens = mod2(phi);
      ]]>
    </sampling_group>

    <sampling_group initial_sample="yes">
      <moments>norm</moments>
      <dependencies>normalisation</dependencies>
      <![CDATA[
        norm = Ncalc;
      ]]>
    </sampling_group>
  </output>
</simulation>
```

We have used the `plan="exhasutive"` option in the `<fftw>` element to ensure that the absolute fastest transform method is found. Because the FFTW package stores the results of its tests (by default in the ~/.xmds/wisdom directory), this option does not cause significant computational overhead, except perhaps on the very first run of a new program.

This simulation introduces the first example of a very powerful feature in XMDS2: the `<computed_vector>` element. This has syntax like any other vector, including possible dependencies on other vectors, and an ability to be used in any element that can use vectors. The difference is that, much like noise vectors, computed vectors are recalculated each time they are required. This means that a computed vector can never be used as an integration vector, as its values are not stored. However, computed vectors allow a simple and efficient method of describing complicated functions of other vectors. Computed vectors may depend on other computed vectors, allowing for spectral filtering and other advanced options. See for example, the *Advanced Topics* section on *Convolutions and Fourier transforms*.

The difference between a computed vector and a stored vector is emphasised by the replacement of the `<initialisation>` element with an `<evaluation>` element. Apart from the name, they have virtually identical purpose and syntax.

```
<computed_vector name="normalisation" dimensions="" type="real">
  <components> Ncalc </components>
  <evaluation>
    <dependencies basis="y">wavefunction</dependencies>
    <![CDATA[
      // Calculate the current normalisation of the wave function.
      Ncalc = mod2(phi);
    ]]>
  </evaluation>
</computed_vector>
```

Here, our computed vector has no transverse dimensions and depends on the components of "wavefunction", so the extra transverse dimensions are integrated out. This code therefore integrates the square modulus of the field, and returns it in the variable "Ncalc". This will be used below to renormalise the "phi" field. Before we examine that process, we have to introduce the `<filter>` element.

The `<filter>` element can be placed in the `<sequence>` element, or inside `<integrate>` elements as we will see next. Elements placed in the `<sequence>` element are executed in the order they are found in the .xmds file. Filter elements place the included CDATA block directly into the generated program at the designated position. If the element does not contain any dependencies, like in our first example, then the code is placed alone:

```
<filter>
  <![CDATA[
    printf("Hello world from a filter segment!\n");
```

```
  ]]>
</filter>
```

This filter block merely prints a string into the output when the generated program is run. If the <filter> element contains dependencies, then the variables defined in those vectors (or computed vectors, or noise vectors) will be available, and the CDATA block will be placed inside loops that run over all the transverse dimensions used by the included vectors. The second filter block in this example depends on both the "wavefunction" and "normalisation" vectors:

```
<filter>
    <dependencies>normalisation wavefunction</dependencies>
  <![CDATA[
    phi *= sqrt(Nparticles/Ncalc);
  ]]>
</filter>
```

Since this filter depends on a vector with the transverse dimension "y", this filter will execute for each point in "y". This code multiplies the value of the field "phi" by the factor required to produce a normalised function in the sense that $\int dy|\phi(y,t)|^2 = N_{particles}$.

The next usage of a <filter> element in this program is inside the <integrate> element, where all filters are placed inside a <filters> element.

```
<filters where="step end">
  <filter>
    <dependencies>wavefunction normalisation</dependencies>
    <![CDATA[
      // Correct normalisation of the wavefunction
      phi *= sqrt(Nparticles/Ncalc);
    ]]>
  </filter>
</filters>
```

Filters placed in an integration block are applied each integration step. The "where" flag is used to determine whether the filter should be applied directly before or directly after each integration step. The default value for the where flag is `where="step start"`, but in this case we chose "step end" to make sure that the final output was normalised after the last integration step.

At the end of the sequence element we introduce the <breakpoint> element. This serves two purposes. The first is a simple matter of convenience. Often when we manage our input and output from a simulation, we are interested solely in storing the exact state of our integration vectors. A breakpoint element does exactly that, storing the components of any vectors contained within, taking all the normal options of the <output> element but not requiring any <sampling_group> elements as that information is assumed.

```
<breakpoint filename="groundstate_break.xsil">
  <dependencies basis="ky">wavefunction</dependencies>
</breakpoint>
```

If the filename argument is omitted, the output filenames are numbered sequentially. Any given <breakpoint> element must only depend on vectors with identical dimensions.

This program begins with a very crude guess to the ground state, but it rapidly converges to the lowest eigenstate.

4.7 Finding the Ground State of a BEC again

Here we repeat the same simulation as in the *Finding the Ground State of a BEC (continuous renormalisation)* example, using a different transform basis. While spectral methods are very effective, and Fourier transforms are typically very efficient due to the Fast Fourier transform algorithm, it is often desirable to describe nonlocal evolution in bases other than the Fourier basis. The previous calculation was the Schrödinger equation with a harmonic potential and a nonlinear term. The eigenstates of such a system are known analytically to be Gaussians

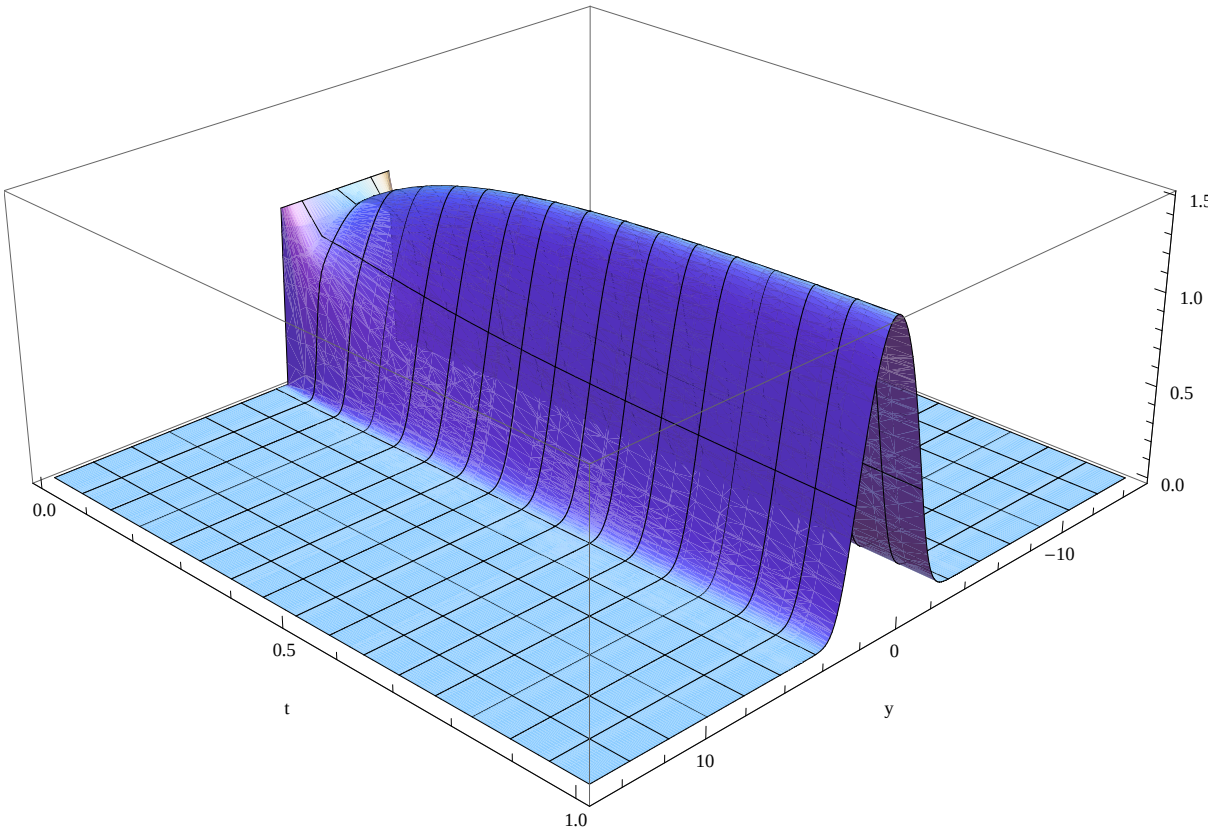


Figure 4.5: The shape of the ground state rapidly approaches the lowest eigenstate. For weak nonlinearities, it is nearly Gaussian.

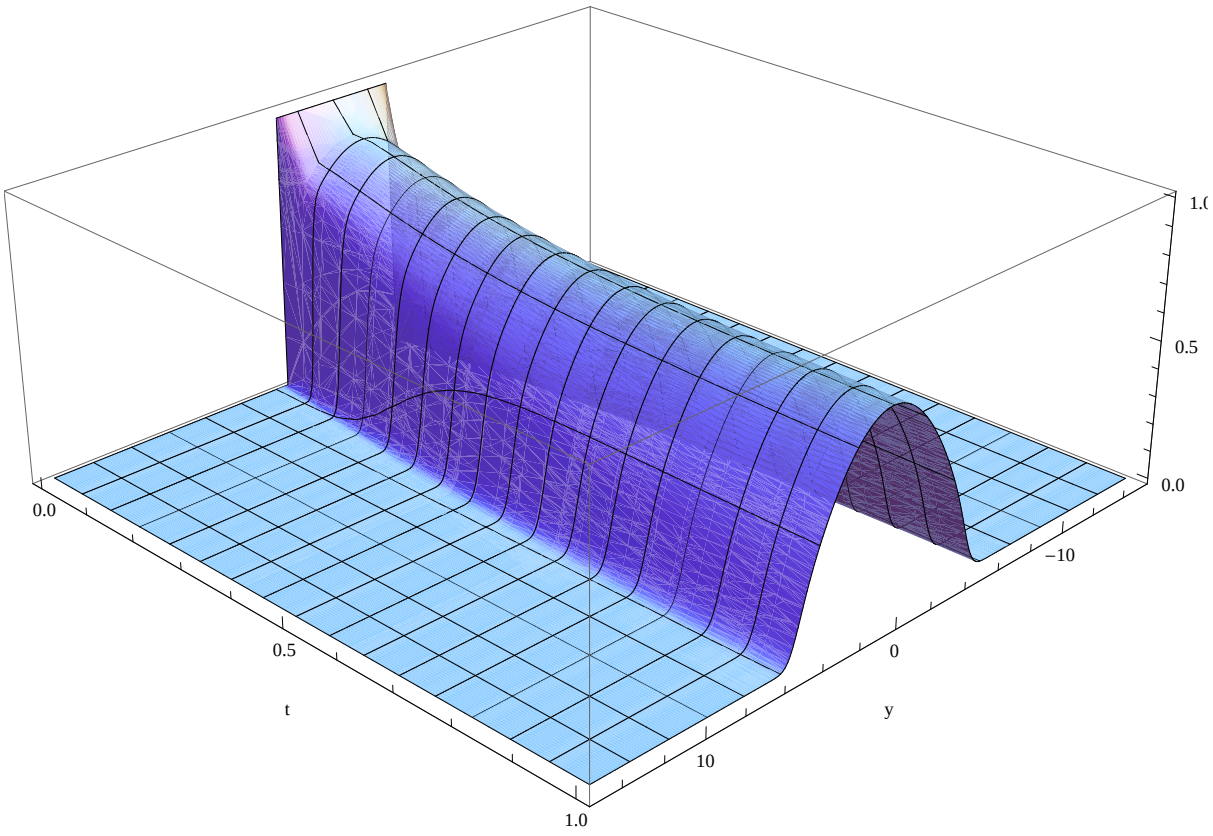


Figure 4.6: When the nonlinear term is larger ($U = 20$), the ground state is wider and more parabolic.

multiplied by the Hermite polynomials.

$$\left[-\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}\omega^2 x^2\right]\phi_n(x) = E_n\phi_n(x)$$

where

$$\phi_n(x,t) = \sqrt{\frac{1}{2^n n!}}\left(\frac{m\omega}{\hbar\pi}\right)^{\frac{1}{4}} e^{-\frac{m\omega x^2}{2\hbar}} H_n\left(\sqrt{\frac{m\omega}{\hbar}}x\right), \qquad E_n = \left(n+\frac{1}{2}\right)\omega$$

where $H_n(u)$ are the physicist's version of the Hermite polynomials. Rather than describing the derivatives as diagonal terms in Fourier space, we therefore have the option of describing the entire $-\frac{\hbar}{2m}\frac{\partial^2}{\partial x^2} + \frac{1}{2}\omega^2 x^2$ term as a diagonal term in the hermite-Gaussian basis. Here is an XMDS2 simulation that performs the integration in this basis. The following is a simplified version of the `examples/hermitegauss_groundstate.xmds` script.

```
<simulation xmds-version="2">
  <name>hermitegauss_groundstate</name>
  <author>Graham Dennis</author>
  <description>
    Solve for the groundstate of the Gross-Pitaevskii equation using the hermite-Gauss basis.
  </description>

  <features>
    <benchmark />
    <bing />
    <validation kind="run-time" />
    <globals>
```

```
    <![CDATA[
      const real omegaz = 2*M_PI*20;
      const real omegarho = 2*M_PI*200;
      const real hbar = 1.05457148e-34;
      const real M = 1.409539200000000e-25;
      const real g = 9.8;
      const real scatteringLength = 5.57e-9;
      const real transverseLength = 1e-5;
      const real Uint = 4.0*M_PI*hbar*hbar*scatteringLength/M/transverseLength/transverseLength
      const real Nparticles = 5.0e5;

      /* offset constants */
      const real EnergyOffset = 0.3*pow(pow(3.0*Nparticles/4*omegarho*Uint,2.0)*M/2.0,1/3.0); /
    ]]>
  </globals>
</features>

<geometry>
  <propagation_dimension> t </propagation_dimension>
  <transverse_dimensions>
    <dimension name="x" lattice="100" length_scale="sqrt(hbar/(M*omegarho))" transform="hermite
  </transverse_dimensions>
</geometry>

<vector name="wavefunction" initial_basis="x" type="complex">
  <components> phi </components>
  <initialisation>
    <![CDATA[
    phi = sqrt(Nparticles) * pow(M*omegarho/(hbar*M_PI), 0.25) * exp(-0.5*(M*omegarho/hbar)*x*x)
    ]]>
  </initialisation>
</vector>

<computed_vector name="normalisation" dimensions="" type="real">
  <components> Ncalc </components>
  <evaluation>
    <dependencies basis="x">wavefunction</dependencies>
    <![CDATA[
      // Calculate the current normalisation of the wave function.
      Ncalc = mod2(phi);
    ]]>
  </evaluation>
</computed_vector>

<sequence>
  <integrate algorithm="ARK45" interval="1.0e-2" steps="4000"  tolerance="1e-10">
    <samples>100 100</samples>
    <filters>
      <filter>
        <dependencies>wavefunction normalisation</dependencies>
        <![CDATA[
          // Correct normalisation of the wavefunction
          phi *= sqrt(Nparticles/Ncalc);
        ]]>
      </filter>
    </filters>
    <operators>
      <operator kind="ip" constant="yes" type="real">
        <operator_names>L</operator_names>
        <![CDATA[
          L = EnergyOffset/hbar - (nx + 0.5)*omegarho;
        ]]>
      </operator>
```

```
        <integration_vectors>wavefunction</integration_vectors>
        <![CDATA[
          dphi_dt = L[phi] - Uint/hbar*mod2(phi)*phi;
        ]]>
      </operators>
    </integrate>

    <filter>
        <dependencies>normalisation wavefunction</dependencies>
      <![CDATA[
        phi *= sqrt(Nparticles/Ncalc);
      ]]>
    </filter>

    <breakpoint filename="hermitegauss_groundstate_break.xsil" format="ascii">
      <dependencies basis="nx">wavefunction</dependencies>
    </breakpoint>
  </sequence>

  <output>
    <sampling_group basis="x" initial_sample="yes">
      <moments>dens</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        dens = mod2(phi);
      ]]>
    </sampling_group>
    <sampling_group basis="kx" initial_sample="yes">
      <moments>dens</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        dens = mod2(phi);
      ]]>
    </sampling_group>
  </output>
</simulation>
```

The major difference in this simulation code, aside from the switch back from dimensionless units, is the new transverse dimension type in the `<geometry>` element.

```
<dimension name="x" lattice="100" length_scale="sqrt(hbar/(M*omegarho))" transform="hermite-gauss
```

We have explicitly defined the "transform" option, which by defaults expects the Fourier transform. The `transform="hermite-gauss"` option requires the 'mpmath' package installed, just as Fourier transforms require the FFTW package to be installed. The "lattice" option details the number of hermite-Gaussian eigenstates to include, and automatically starts from the zeroth order polynomial and increases. The number of hermite-Gaussian modes fully determines the irregular spatial grid up to an overall scale given by the `length_scale` parameter.

The `length_scale="sqrt(hbar/(M*omegarho))"` option requires a real number, but since this script defines it in terms of variables, XMDS2 is unable to verify that the resulting function is real-valued at the time of generating the code. XMDS2 will therefore fail to compile this program without the feature:

```
<validation kind="run-time" />
```

which disables many of these checks at the time of writing the C-code.

4.8 Multi-component Schrödinger equation

This example demonstrates a simple method for doing matrix calculations in XMDS2. We are solving the multi-component PDE

$$\frac{\partial \phi_j(x,y)}{\partial t} = \frac{i}{2}\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right)\phi_j(x,y) - iU(x,y)\sum_k V_{jk}\phi_k(x,y)$$

where the last term is more commonly written as a matrix multiplication. Writing this term out explicitly is feasible for a small number of components, but when the number of components becomes large, or perhaps V_{jk} should be precomputed for efficiency reasons, it is useful to be able to perform this sum over the integer dimensions automatically. This example show how this can be done naturally using a computed vector. The XMDS2 script is as follows:

```
<simulation xmds-version="2">
  <name>2DMSse</name>

  <author>Joe Hope</author>
  <description>
    Schroedinger equation for multiple internal states in two spatial dimensions.
  </description>

  <features>
      <benchmark />
      <bing />
      <fftw plan="patient" />
      <openmp />
      <auto_vectorise />
     </features>

  <geometry>
      <propagation_dimension> t </propagation_dimension>
      <transverse_dimensions>
          <dimension name="x" lattice="32"  domain="(-6, 6)" />
          <dimension name="y" lattice="32"  domain="(-6, 6)" />
          <dimension name="j" type="integer" lattice="2" domain="(0,1)" aliases="k"/>
      </transverse_dimensions>
   </geometry>

  <vector name="wavefunction" type="complex" dimensions="x y j">
    <components> phi </components>
    <initialisation>
      <![CDATA[
      phi = j*sqrt(2/sqrt(M_PI/2))*exp(-(x*x+y*y)/4)*exp(i*0.1*x);
      ]]>
    </initialisation>
  </vector>

  <vector name="spatialInteraction" type="real" dimensions="x y">
    <components> U </components>
    <initialisation>
      <![CDATA[
      U=exp(-(x*x+y*y)/4);
      ]]>
    </initialisation>
  </vector>

  <vector name="internalInteraction" type="real" dimensions="j k">
    <components> V </components>
    <initialisation>
      <![CDATA[
```

```
      V=3*(j*(1-k)+(1-j)*k);
      ]]>
    </initialisation>
  </vector>

  <computed_vector name="coupling" dimensions="x y j" type="complex">
    <components>
      VPhi
    </components>
    <evaluation>
      <dependencies basis="x y j k">internalInteraction wavefunction</dependencies>
      <![CDATA[
        // Calculate the current normalisation of the wave function.
        VPhi = V*phi(j => k);
      ]]>
    </evaluation>
  </computed_vector>

  <sequence>
    <integrate algorithm="ARK45" interval="2.0" tolerance="1e-7">
      <samples>20 100</samples>
      <operators>
        <integration_vectors>wavefunction</integration_vectors>
        <operator kind="ex" constant="yes">
          <operator_names>Ltt</operator_names>
          <![CDATA[
            Ltt = -i*(kx*kx+ky*ky)*0.5;
          ]]>
        </operator>
        <![CDATA[
        dphi_dt = Ltt[phi] -i*U*VPhi;
        ]]>
        <dependencies>spatialInteraction coupling</dependencies>
      </operators>
    </integrate>
  </sequence>

  <output>
    <sampling_group basis="x y j" initial_sample="yes">
      <moments>density</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        density = mod2(phi);
      ]]>
    </sampling_group>
    <sampling_group basis="x(0) y(0) j" initial_sample="yes">
      <moments>normalisation</moments>
      <dependencies>wavefunction</dependencies>
      <![CDATA[
        normalisation = mod2(phi);
      ]]>
    </sampling_group>
  </output>
</simulation>
```

The only truly new feature in this script is the "aliases" option on a dimension. The integer-valued dimension in this script indexes the components of the PDE (in this case only two). The V_{jk} term is required to be a square array of dimension of this number of components. If we wrote the k-index of V_{jk} using a separate `<dimension>` element, then we would not be enforcing the requirement that the matrix be square. Instead, we note that we will be using multiple 'copies' of the j-dimension by using the "aliases" tag.

```
<dimension name="j" type="integer" lattice="2" domain="(0,1)" aliases="k"/>
```

This means that we can use the index "k", which will have exactly the same properties as the "j" index. This is used to define the "V" function in the "internalInteraction" vector. Now, just as we use a computed vector to perform an integration over our fields, we use a computed vector to calculate the sum.

```
<computed_vector name="coupling" dimensions="x y j" type="complex">
  <components>
    VPhi
  </components>
  <evaluation>
    <dependencies basis="x y j k">internalInteraction wavefunction</dependencies>
    <![CDATA[
      // Calculate the current normalisation of the wave function.
      VPhi = V*phi(j => k);
    ]]>
  </evaluation>
</computed_vector>
```

Since the output dimensions of the computed vector do not include a "k" index, this index is integrated. The volume element for this summation is the spacing between neighbouring values of "j", and since this spacing is one, this integration is just a sum over k, as required.

By this point, we have introduced most of the important features in XMDS2. More details on other transform options and rarely used features can be found in the *Advanced Topics* section.

CHAPTER

FIVE

REFERENCE SECTION

Contents:

5.1 Configuration, installation and runtime options

Running the 'xmds2' program with the option '–help', gives several options that can change its behaviour at runtime. These

- '-o' or '–output', which overrides the name of the output file to be generated
- '-n' or '–no-compile', which generates the C code for the simulation, but does not try to compile it
- '-v' or '–verbose', which gives verbose output about compilation flags.
- '-g' or '–debug', which compiles the simulation in debug mode (compilation errors refer to lines in the source, not the .xmds file). This option implies '-v'. This option is mostly useful when debugging XMDS code generation.
- '–waf-verbose', which makes `waf` be very verbose when configuring XMDS or compiling simulations. This option is intended for developer use only to aid in diagnosing problems with `waf`.

It also has commands to configure XMDS2 and recheck the installation. If your program requires extra paths to compile, you can configure XMDS2 to include those paths by default. Simply use the command

```
$ xmds2 --configure --include-path /path/to/include --lib-path /path/to/lib
```

Alternatively, you can set the `CXXFLAGS` or `LINKFLAGS` environment variables before calling `xmds2 --reconfigure`. For example, to pass the compiler flag `-pedantic` and the link flag `-lm` using the bash shell, use:

```
$ export CXXFLAGS="-pedantic"
$ export LINKFLAGS="-lm"
$ xmds2 --reconfigure ''
```

This method can also be used to change the default compilers for standard and parallel processing, using the CXX and MPICXX flags respectively.

Running XMDS2 with the '–configure' option also searches for packages that have been installed since you last installed or configured XMDS2. If you wish to run 'xmds2 –configure' with the same extra options as last time, simply use the command:

```
$ xmds2 --reconfigure
```

A detailed log of the checks is saved in the file '~/.xmds/waf_configure/config.log'. This can be used to identify issues with packages that XMDS2 is not recognised, but you think that you have successfully installed on your system.

5.2 Useful XML Syntax

Standard XML placeholders can be used to simplify some scripts. For example, the following (abbreviated) code ensures that the limits of a domain are symmetric.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE simulation [
<!ENTITY Npts     "64">
<!ENTITY L      "3.0e-5">
]>
  <simulation xmds-version="2">

    . . .

    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
          <dimension name="x" lattice="&Npts;"  domain="(-&L;, &L;)" />
        </transverse_dimensions>
     </geometry>
```

5.3 XMDS2 XML Schema

There are many, many XML tags that can make up an XMDS2 script. Most of them are optional, or have default values if not specified. It is, however, useful to know which elements are possible, and their position and relationship to other elements in the script. Shown below is the full XML tree for XMDS2, which shows all possible elements and their position in the tree. An ellipsis (...) outside an element indicates the element above can be repeated indefinitely, and an ellipsis inside an element (<element> ... </element>) indicates that the structure of that element has already been shown previously.

The syntax <element /> can be used for lowest-level elements that have attributes but no content, and are shorthand for <element> </element>. This shorthand notation can also be used for elements which can only contain the content "yes" or "no"; in this case the presence of <element /> is equivalent to <element> yes </element>, and the absence of such an element is equivalent to <element> no </element>

The possible attributes and attribute values for each element are not shown; see the individual entries in the Reference section for details.

```
<?xml version="1.0" encoding="UTF-8"?>
<simulation xmds-version="2">
  <name> </name>
  <author> <author>
  <description> </description>

  <features>
    <arguments>
      <argument />
      <argument />
      ...
    </arguments>
    <auto_vectorise />
    <benchmark />
    <bing />
    <cflags> </cflags>
    <chunked_output />
    <diagnostics />
    <error_check />
    <halt_non_finite />
    <fftw />
    <globals> </globals>
```

```
  <openmp />
  <precision> </precision>
  <validation />
</features>

<driver />

<geometry>
  <propagation_dimension> </propagation_dimension>
  <transverse_dimensions>
    <dimension />
    <dimension />
    ...
  </transverse_dimensions>
</geometry>

<vector>
  <components> </components>
  <initialisation>
    <dependencies> </dependencies>
    <filename>
    <![CDATA [
    ]]>
  </initialisation>
</vector>

<vector> ... </vector>
<vector> ... </vector>
...

<computed_vector>
  <components> </components>
  <evaluation>
    <dependencies> </dependencies>
    <![CDATA [
    ]]>
  </evaluation>
</computed_vector>

<computed_vector> ... </computed_vector>
<computed_vector> ... </computed_vector>
...

<noise_vector>
  <components> </components>
</noise_vector>

<noise_vector> ... </noise_vector>
<noise_vector> ... </noise_vector>
...

<sequence>

  <filter>
    <dependencies> </dependencies>
    <![CDATA [
    ]]>
  </filter>

  <integrate>
    <samples> </samples>

    <computed_vector> ... </computed_vector>
```

```
      <filters>
        <filter> ... </filter>
        <filter> ... </filter>
        ...
      </filters>

      <operators>

        <operator>
          <boundary_condition>
            <dependencies> </dependencies>
            <![CDATA [
            ]]>
          </boundary_condition>
          <operator_names> </operator_names>
          <dependencies> </dependencies>
          <![CDATA [
          ]]>
        </operator>

        <operator> ... </operator>
        <operator> ... </operator>
        ...

        <integration_vectors> </integration_vectors>
        <dependencies> </dependencies>
        <![CDATA [
        ]]>

      </operators>

    </integrate>

    <breakpoint>
      <dependencies> </dependencies>
    </breakpoint>

  </sequence>

  <output>
    <sampling_group>
      <dependencies> </dependencies>
      <moments> </moments>
      <operator> ... </operator>
      <![CDATA [
      ]]>
    </sampling_group>

    <sampling_group> ... </sampling_group>
    <sampling_group> ... </sampling_group>
    ...

  </output>

</simulation>
```

5.4 XMDS2 script elements

This section outlines all the elements and options available in an XMDS2 script. This is very much a **work in progress**, beginning with placeholders in most cases, as we have prioritised the tutorials for new users. One

of the most productive ways that non-developer veterans can contribute to the project is to help develop this documentation.

5.4.1 Simulation element

The `<simulation>` element is the single top level element in an XMDS2 simulation, and contains all the other elements. All XMDS scripts must contain exactly one simulation element, and it must have the `xmds-version="2"` attribute defined.

Example syntax:

```
<simulation xmds-version="2">
    <!-- Rest of simulation goes here -->
</simulation>
```

5.4.2 Name element

The name of your simulation. This element is optional, but recommended. If it is set, it will be the name of the executable file generated from this script. It will also be the name of the output file (with an appropriate extension) if the `filename` attribute is not given a value in the `<output>` element.

Example syntax:

```
<name> funky_solver </name>
```

5.4.3 Author element

The author(s) of this script. This element is optional, but can be useful if you need to find the person who has written an incomprehensible script and thinks comments are for the weak.

Example syntax:

```
<author> Ima Mollusc </author>
```

5.4.4 Description element

A description of what the simulation does. Optional, but recommended, in case you (or someone else) has to revist the script at some distant point in the future.

Example syntax:

```
<description>
  Calculate the 3D ground state of a Rubidium BEC in a harmonic magnetic trap assuming
  cylindrical symmetry about the z axis and reflection symmetry about z=0.
  This permits us to use the cylindrical Bessel functions to expand the solution transverse
  to z and a cosine series to expand the solution along z.
</description>
```

5.4.5 Features Elements

Features elements are where simulation-wide options are specified. The `<features>` element wraps one or more elements describing features. There are many possible feature elements. Currently, a full list of the features supported is:

- *arguments*
- *auto_vectorise*

- *benchmark*
- *bing*
- *cflags*
- *chunked_output*
- *diagnostics*
- *error_check*
- *halt_non_finite*
- *fftw*
- *globals*
- *OpenMP*
- *precision*
- *validation*

Example syntax:

```
<simulation xmds-version="2">
  <features>
    <bing />
    <precision> double </precision>
    ...
  </features>
</simulation>
```

Arguments Element

The `<arguments>` element is optional, and allows defining variables that can be passed to the simulation at run time. These variables are then globally accessible throughout the simulation script. Each of the variables must be defined in an `<argument>` element (see below). The variables can then be passed to the simulation executable as options on the command line. For example, one could define the variables `size`, `number`, and `pulse_shape`

```
<name> arguments_test </name>
<features>
  <arguments>
    <argument name="size" type="real" default_value="20.0"/>
    <argument name="number" type="integer" default_value="7"/>
    <argument name="pulse_shape" type="string" default_value="gaussian"/>
  </arguments>
</features>
```

When `XMDS2` is run on this script the executable `arguments_test` is created. The values of `size`, `number`, and `pulse_shape` can then be set to whatever is desired at runtime via

```
./arguments_test --size=1.3 --number=2 --pulse_shape=lorentzian
```

It is also possible to include an optional `CDATA` block inside the `<arguments>` block. This code will run after the arguments have been initialised with the values passed from the command line. This code block could be used, for example, to sanity check the parameters passed in, or for assigning values to global variables based on those parameters. Any references to variables defined in an `<argument>` element should be made here rather than in the *Globals* element, or else the variables will only have their default values. For example, one could have the following

```
<features>
  <globals>
    <![CDATA[
      real atom_kick;
```

```
    ]]>
  <globals>
  <arguments>
    <argument name="bragg_order" type="integer" default_value="2"/>
    <![CDATA[
      atom_kick = bragg_order * 2*M_PI / 780e-9;
    ]]>
  </arguments>
</features>
```

Argument element

Each <argument> element describes one variable that can be passed to the simulation at runtime via the command line. There are three mandatory attributes: name, type, and default_value. name is the name by which you can refer to that variable later in the script, as well as the name of the command line parameter. type defines the data type of the variable, and default_value is the value to which the variable is set if it is not given a value on the command line.

Auto_vectorise element

The <auto_vectorise /> feature attempts to activate automatic vectorisation for large loops, if it is available in the compiler. This should make some simulations go faster.

Benchmark

The <benchmark /> feature includes a timing routine in the generated code, so that it is possible to see how long the simulations take to run.

Bing

The <bing /> feature causes the simulation to make an invigorating sound when the simulation finishes executing.

C Flags

The <cflags> feature allows extra flags to be passed to the compiler. This can be useful for optimisation, and also using specific external libraries. The extra options to be passed are defined with a 'CDATA' block. The compile options can be made visible by running XMDS2 either with the "-v" (verbose) option, or the "-g" (debug) option.

Example syntax:

```
<cflags>
    <![CDATA[
        -O4
    ]]>
</cflags>
```

Chunked Output

By default, XMDS2 keeps the contents of all output moment groups in memory until the end of the simulation when they are written to the output file. This can be a problem if your simulation creates a very large amount of output. <chunked_output /> causes the simulation to save the output data in chunks as the simulation progresses. For some simulations this can significantly reduce the amount of memory required. The amount of data in a chunk can be specified with the size attribute where the suffixes "KB" (kilobytes), "MB" (megabytes), "GB"

(gigabytes) and "TB" (terabytes) are understood. Note that `size` specifies the chunk size per output sampling group, per MPI process. So a chunk size of 4MB for a distributed-MPI simulation using 20 processes will cause each process to save up 4MB of data, and data to be written to the output file 80MB at a time.

Limitations (XMDS will give you an error if you violate any of these):

- This feature cannot be used with the ASCII output file format due to limitations in the file format.
- This feature cannot be used with the `multi-path` drivers because all sampling data is required to compute the mean and standard error statistics.
- Neither is this feature compatible with the `error_check` feature as that relies on all sampling data being available to compute the error.

Example syntax:

```
<simulation xmds-version="2">
    <features>
        <chunked_output size="5MB" />
    </features>
</simulation>
```

Diagnostics

The `<diagnostics />` feature causes a simulation to output more information as it executes. This should be useful when a simulation is dying / giving bad results to help diagnose the cause. Currently, it largely outputs step error information.

Error Check

It's often important to know whether you've got errors. This feature runs each integration twice: once with the specified error tolerance or defined lattice spacing in the propagation dimension, and then again with half the lattice spacing, or an equivalently lower error tolerance. Each component of the output then shows the difference between these two integrations as an estimate of the error. This feature is particularly useful when integrating stochastic equations, as it treats the noise generation correctly between the two runs, and thus makes a reasonable estimate of the strong convergence of the equations.

Example syntax:

```
<simulation xmds-version="2">
    <features>
        <error_check />
    </features>
</simulation>
```

Halt_Non_Finite

The `<halt_non_finite />` feature is used to stop computations from continuing to run after the vectors stop having numerical values. This can occur when a number is too large to represent numerically, or when an illegal operation occurs. Processing variables with non-numerical values is usually much slower than normal processing, and the results are meaningless. Of course, there is a small cost to introducing a run-time check, so this feature is optional.

fftw element

The `<fftw \>` feature can be used to pass options to the Fast Fourier Transform library used by XMDS. This library tests algorithms on each architecture to determine the fastest method of solving each problem. Typically this costs very little overhead, as the results of all previous tests are stored in the directory "~/.xmds/wisdom". The level of detail for the search can be specified using the `plan` attribute, which can take values of `"estimate"`,

"measure",‘”patient”, or ``"exhaustive", in order of the depth of the search. The number of threads for threaded FFTs can be specified with the threads attribute, which must be a positive integer.

Example syntax:

```
<fftw plan="patient" threads="3" />
```

Globals

The globals feature places the contents of a ‘CDATA’ block near the top of the generated program. Amongst other things, this is useful for defining variables that are then accessible throughout the entire program.

Example syntax:

```
<globals>
  <![CDATA[
    const real omegaz = 2*M_PI*20;
    long Nparticles = 50000;

    /* offset constants */
    real frequency = omegaz/2/M_PI;
  ]]>
</globals>
```

OpenMP

The <openmp /> feature instructs compatible compilers to parallelise key loops using the OpenMP API standard. By default the simulation will use all available CPUs. The number of threads used can be restricted by specifying the number of threads in the script with <openmp threads="2"/>, or by setting the OMP_NUM_THREADS environment variable at run-time like so:

```
OMP_NUM_THREADS=2 ./simulation_name
```

Precision

This specifies the precision of the XMDS2 real and complex datatypes, as well as the precision used when computing transforms. Currently two values are accepted: single and double. If this feature isn't specified, XMDS2 defaults to using double precision for its variables and internal calculations.

Single precision has approximately 7.2 decimal digits of accuracy, with a minimum value of 1.4×10^{-45} and a maximum of 3.8×10^{34}. Double precision has approximately 16 decimal digits of accuracy, a minimum value of 4.9×10^{-324} and a maximum value of 1.8×10^{308}.

Using single precision can be attractive, as it can be more than twice as fast, depending on whether a simulation is CPU bound, memory bandwidth bound, MPI bound or bottlenecked elsewhere, although in some situations you may see no speed-up at all. Caution should be exercised, however. Keep in mind how many timesteps your simulation requires, and take note of the tolerance you have set per step, to see if the result will lie within your acceptable total error - seven digit precision isn't a lot. Quite apart from the precision, the range of single precision can often be inadequate for many physical problems. In atomic physics, for example, intermediate values below 1.4×10^{-45} are easily obtained, and will be taken as zero. Similarly, values above 3.8×10^{34} will result in NaNs and make the simulation results invalid.

Also note that when using an adaptive step integrator, setting a tolerance close to limits of the precision can lead to very slow performance.

A further limitation is that not all the combinations of random number generators and probability distributions that are supported in double precision are supported in single precision. For example, the solirte generator does not support single precision gaussian distributions. dsfmt, however, is one of the fastest generators, and does support single precision.

WARNING: Single precision mode has not been tested anywhere near as thoroughly as the default double precision mode, and there is a higher chance you will run into bugs.

Example syntax:

```
<simulation xmds-version="2">
    <features>
        <precision> single </precision>
    </features>
</simulation>
```

Validation

XMDS2 makes a large number of checks in the code generation process to verify that the values for all parameters are safe choices. Sometimes we wish to allow these parameters to be specified by variables. This opens up many possibilities, but requires that any safety checks for parameters be performed during the execution of the program itself. The <validation> feature activates that option, with allowable attributes being "run-time", "compile-time" and "none".

As an example, one may wish to define the number of grid points and the range of the grid at run-time rather than explicitly define them in the XMDS2 script. To accomplish this, one could do the following:

```
 <name> validation_test </name>
 <features>
   <validation kind="run-time" />
   <arguments>
     <argument name="xmin" type="real" default_value="-1.0"/>
     <argument name="xmax" type="real" default_value="1.0"/>
     <argument name="numGridPoints" type="integer" default_value="128"/>
   </arguments>
 </features>

 <geometry>
   <propagation_dimension> t </propagation_dimension>
   <transverse_dimensions>
     <dimension name="x" lattice="numGridPoints"  domain="(xmin, xmax)" />
   </transverse_dimensions>
</geometry>
```

and then run the resulting executable with:

```
./validation_test --xmin=-2.0 --xmax=2.0 --numGridPoints=64
```

This approach means that when XMDS2 is parsing the script it is unable to tell, for example, if the number of sampling points requested is less than or equal to the lattice size. Consequently it will create an executable with "numGridPoints" as an internal variable, and make the check at run-time, when it knows the value of "numGridPoints" rather than at compile time, when it doesn't.

5.4.6 Driver Element

The driver element controls the overall management of the simulation, including how many paths of a stochastic simulation are to be averaged, and whether or not it is to be run using distributed memory parallelisation. If it is not included, then the simulation is performed once without using MPI parallelisation. If it is included, it must have a `name` attribute.

The `name` attribute can have values of "none" (which is equivalent to the default option of not specifying a driver), "distributed-mpi", "multi-path" or "mpi-multi-path".

Choosing the `name="distributed-mpi"` option allows a single integration over multiple processors. The resulting executable can then be run according to your particular implementation of MPI. The FFTW library only allows MPI processing of multidimensional vectors, as otherwise shared memory parallel processing requires too much inter-process communication to be efficient. Maximally efficient parallelisation occurs where evolution is

entirely local in one transverse dimension (see *transverse dimensions* below). In that case, that dimension should be listed first in the *<geometry>* element. As noted in the worked example *Wigner Function*, it is wise to test the speed of the simulation using different numbers of processors.

The `name="multi-path"` option is used for stochastic simulations, which are typically run multiple times and averaged. It requires a `paths` attribute with the number of iterations of the integration to be averaged. The output will report the averages of the desired samples, and the standard error in those averages. The `name="mpi-multi-path"` option integrates separate paths on different processors, which is typically a highly efficient process.

Example syntax:

```
<simulation xmds-version="2">
    <driver name="distributed-mpi" />
        <!-- or -->
    <driver name="multi-path" paths="10" />
        <!-- or -->
    <driver name="mpi-multi-path" paths="1000" />
</simulation>
```

5.4.7 Geometry Element

The `<geometry>` element describes the dimensions used in your simulation, and is required. The only required element inside is the `<propagation_dimension>` element, which defines the name of the dimension along which your simulation will integrate. Nothing else about this dimension is specified, as requirements for the lattice along the integration dimension is specified by the `<integrate>` blocks themselves, as described in section *Integrate element*. If there are other dimensions in your problem, they are called "transverse dimensions", and are described in the `<transverse_dimensions>` element. Each dimension is then described in its own `<dimension>` element. A transverse dimension must have a unique name defined by a `name` attribute. If it is not specified, the type of dimension will default to "real", otherwise it can be specified with the `type` attribute. Allowable types (other than "real") are "long", "int", and "integer", which are actually all synonyms for an integer-valued dimension.

Each transverse dimension must specify how many points or modes it requires, and the range over which it is defined. This is done by the `lattice` and `domain` attributes respectively. The `lattice` attribute is an integer, and is optional for integer dimensions, where it can be defined implicitly by the domain. The `domain` attribute is specified as a pair of numbers (e.g. `domain="(-17,3)"`) defining the minimum and maximum of the grid.

Any dimension can have a number of aliases. These act exactly like copies of that dimension, but must be included explicitly in the definition of subsequent vectors (i.e. they are not included in the default list of dimensions for a new vector). The list of aliases for a dimension are included in an `aliases` attribute. They are useful for non-local reference of variables. See `groundstate_gaussian.xmds` and `2DMultistateSE.xmds` as examples.

Integrals over a dimension can be multiplied by a common prefactor, which is specified using the `volume_prefactor` attribute. For example, this allows the automatic inclusion of a factor of two due to a reflection symmetry by adding the attribute `volume_prefactor="2"`. In very specific cases, you may wish to refer to volume elements explicitly. This will lead to grid-dependent behaviour, which is sometimes required in certain stochastic field simulations, for example. In this case, the volume element for each variable is described by a `d` prefix (e.g. `lambda` would be referred to as `dlambda`). These volume elements contain any implicit prefactors (for example, the radial coordinate for dimensions defined using *Bessel transforms*), including the `volume_prefactor` element.

If you are using the `distributed-mpi` driver to parallelise the simulation, place the dimension you wish to split over multiple processors first. The most efficient parallelisation would involve distributing a dimension with only local evolution, as the different memory blocks would not need to communicate. Nonlocal evolution that is local in Fourier space is the second preference, as the Fourier transform can also be successfully parallelised with minimum communication. Each transverse dimension can be associated with a transform. This allows the simulation to manipulate vectors defined on that dimension in the transform space. The default is Fourier space (with the associated transform being the discrete Fourier transform, or "dft"), but others can be specified with

the `transform` attribute. The other options are "none", "dst", "dct", "bessel", "spherical-bessel" and "hermite-gauss". Using the right transform can dramatically improve the speed of a calculation.

An advanced feature discussed further in *Dimension aliases* are dimension aliases, which are specified by the `aliases` attribute. This feature is useful for example, when calculating correlation functions.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <!-- A real-valued dimension from -1.5 to 1.5 -->
            <dimension name="x" lattice="128" domain="(-1.5, 1.5)" />

            <!-- An integer-valued dimension with the 6 values -2, -1, 0, 1, 2, 3 -->
            <dimension name="j"                domain="(-2,3)" type="integer" />

            <!-- A real-valued dimension using the bessel transform for a radial coordinate -->
            <dimension name="r" lattice="64" domain="(0, 5)"  transform="bessel" volume_prefactor=
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "dft" transform

The "dft" transform is performed using the the normal discrete Fourier transform, which means that it enforces periodic boundary conditions on vectors defined on that dimension. Another implication is that it can only be used with complex-valued vectors. The discrete Fourier transform is almost exactly the same as a standard Fourier transform. The standard Fourier transform is

$$\mathcal{F}\left[f(x)\right](k) = \frac{1}{2\pi}\int_{x_{\min}}^{x_{\max}} f(x)e^{-ikx}dx$$

The discrete Fourier transform has no information about the domain of the lattice, so the XMDS2 transform is equivalent to

$$\begin{aligned}\tilde{\mathcal{F}}\left[f(x)\right](k) &= \frac{1}{2\pi}\int_{x_{\min}}^{x_{\max}} f(x)e^{-ik(x+x_{\min})}dx \\ &= e^{-ix_{\min}k}\mathcal{F}\left[f(x)\right](k)\end{aligned}$$

The standard usage in an XMDS simulation involves moving to Fourier space, applying a transformation, and then moving back. For this purpose, the two transformations are entirely equivalent as the extra phase factor cancels. However, when fields are explicitly defined in Fourier space, care must be taken to include this phase factor explicitly. See section *Convolutions and Fourier transforms* in the Advanced Topics section.

When a dimension uses the "dft" transform, then the Fourier space variable is defined as the name of the dimension prefixed with a "k". For example, the dimensions "x", "y", "z" and "tau" will be referenced in Fourier space as "kx","ky", "kz" and "ktau".

Fourier transforms allow easy calculation of derivatives, as the n^{th} derivative of a field is proportional to the n^{th} moment of the field in Fourier space:

$$\mathcal{F}\left[\frac{\partial^n f(x)}{\partial x^n}\right](k_x) = (i\,k_x)^n\,\mathcal{F}\left[f(x)\right](k_x)$$

This identity can be used to write the differential operator $\mathcal{L} = \frac{\partial}{\partial x}$ as an `IP` or `EX` operator as `L = i*kx;` (see *Operators and operator elements* for more details).

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <!-- transform="dft" is the default, omitting it wouldn't change anything -->
            <dimension name="x" lattice="128" domain="(-1.5, 1.5)" transform="dft" />
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "dct" transform

The "dct" (discrete cosine transform) is a Fourier-based transform that implies different boundary conditions for associated vectors. XMDS uses the type-II DCT, often called "the DCT", and its inverse, which is also called the type-III DCT. This transform assumes that any vector using this dimension is both periodic, and also even around a specific point within each period. The grid is therefore only defined across a half period in order to sample each unique point once, and can therefore be of any shape where all the odd derivatives are zero at each boundary. This is a very different boundary condition compared to the DFT, which demands periodic boundary conditions, and is therefore suitable for different simulations. For example, the DCT is a natural choice when implementing zero Neumann boundary conditions.

As the DCT transform can be defined on real data rather only complex data, it can also be superior to DFT-based spectral methods for simulations of real-valued fields where boundary conditions are artificial.

XMDS labels the cosine transform space variables the same as for *Fourier transforms* and all the even derivatives can be calculated the same way. Odd moments of the cosine-space variables are in fact *not* related to the corresponding odd derivatives by an inverse cosine transform.

Discrete cosine transforms allow easy calculation of even-order derivatives, as the $2n^{\text{th}}$ derivative of a field is proportional to the $2n^{\text{th}}$ moment of the field in DCT-space:

$$\mathcal{F}_{\text{DCT}}\left[\frac{\partial^{2n} f(x)}{\partial x^{2n}}\right](k_x) = (-k_x^2)^n \, \mathcal{F}_{\text{DCT}}\left[f(x)\right](k_x)$$

This identity can be used to write the differential operator $\mathcal{L} = \frac{\partial^2}{\partial x^2}$ as an `IP` or `EX` operator as `L = -kx*kx;` (see *Operators and operator elements* for more details).

For problems where you are defining the simulation domain over only half of the physical domain to take advantage of reflection symmetry, consider using `volume_prefactor="2.0"` so that all volume integrals are over the entire physical domain, not just the simulation domain. i.e. integrals would be over -1 to 1 instead of 0 to 1 if the domain was specified as `domain="(0,1)"`.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(-1.5, 1.5)" transform="dct" />
                <!-- Or to cause volume integrals to be multiplied by 2 -->
            <dimension name="y" lattice="128" domain="(0, 1)" transform="dct" volume_prefactor="2
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "dst" transform

The "dst" (discrete sine transform) is a counterpart to the DCT transform. XMDS uses the type-II DST and its inverse, which is also called the type-III DST. This transform assumes that fields are periodic in this dimension,

but also that they are also odd around a specific point within each period. The grid is therefore only defined across a half period in order to sample each unique point once, and can therefore be of any shape where all the even derivatives are zero at each boundary.

The DST transform can be defined on real-valued vectors. As odd-valued functions are zero at the boundaries, this is a natural transform to use when implementing zero Dirichlet boundary conditions.

XMDS labels the sine transform space variables the same as for *Fourier transforms* and all the even derivatives can be calculated the same way. Odd moments of the sine-space variables are in fact *not* related to the corresponding odd derivatives by an inverse sine transform.

Discrete sine transforms allow easy calculation of even-order derivatives, as the $2n^{\text{th}}$ derivative of a field is proportional to the $2n^{\text{th}}$ moment of the field in DST-space:

$$\mathcal{F}_{\text{DST}}\left[\frac{\partial^{2n} f(x)}{\partial x^{2n}}\right](k_x) = (-k_x^2)^n \, \mathcal{F}_{\text{DST}}\left[f(x)\right](k_x)$$

This identity can be used to write the differential operator $\mathcal{L} = \frac{\partial^2}{\partial x^2}$ as an `IP` or `EX` operator as `L = -kx*kx;` (see *Operators and operator elements* for more details).

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(0, 1.5)" transform="dst" />
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "bessel" transform

Just as the Fourier basis is useful for finding derivatives in Euclidean geometry, the basis of Bessel functions is useful for finding certain common operators in cylindrical co-ordinates. In particular, we use the Bessel functions of the first kind, $J_m(u)$. The relevant transform is the Hankel transform:

$$F_m(k) = \mathcal{H}_m\left[f\right](k) = \int_0^\infty r f(r) J_m(kr) dr$$

which has the inverse transform:

$$f(r) = \mathcal{H}_m^{-1}\left[F_m\right](r) = \int_0^\infty k F_m(k) J_m(kr) dk$$

This transform pair has the useful property that the Laplacian in cylindrical co-ordinates is diagonal in this basis:

$$\nabla^2\left(f(r)e^{im\theta}\right) = \left(\frac{\partial^2 f}{\partial r^2} + \frac{1}{r}\frac{\partial f}{\partial r} - \frac{m^2}{r^2} f\right) e^{im\theta} = \left\{\mathcal{H}_m^{-1}\left[(-k^2)F_m(k)\right](r)\right\} e^{im\theta}$$

XMDS labels the variables in the transformed space with a prefix of 'k', just as for *Fourier transforms*. The order m of the transform is defined by the `order` attribute in the `<dimension>` element, which must be assigned as a non-negative integer. If the order is not specified, it defaults to zero which corresponds to the solution being independent of the angular coordinate θ.

It can often be useful to have a different sampling in normal space and Hankel space. Reducing the number of modes in either space dramatically speeds simulations. To set the number of lattice points in Hankel space to be

different to the number of lattice points for the field in its original space, use the attribute `spectral_lattice`. The Bessel space lattice is chosen such that the boundary condition at the edge of the domain is zero. This ensures that all of the Bessel modes are orthogonal. The spatial lattice is also chosen in a non-uniform manner so that Gaussian quadrature methods can be usedfor spectrally accurate transforms.

Hankel transforms allow easy calculation of the Laplacian of fields with cylindrical symmetry. Applying the operator `L = -kr*kr` in Hankel space is therefore equivalent to applying the operator

$$\mathcal{L} = \left(\frac{\partial^2}{\partial r^2} + \frac{1}{r}\frac{\partial}{\partial r} - \frac{m^2}{r^2} \right)$$

in coordinate space.

In non-Euclidean co-ordinates, integrals have non-unit volume elements. For example, in cylindrical co-ordinates with a radial co-ordinate 'r', integrals over this dimension have a volume element rdr. When performing integrals along a dimension specified by the "bessel" transform, the factor of the radius is included implicitly. If you are using a geometry with some symmetry, it is common to have prefactors in your integration. For example, for a two-dimensional volume in cylindrical symmetry, all integrals would have a volume element of $2\pi rdr$. This extra factor of 2π can be included for all integrals by specifying the attribute `volume_prefactor="2*M_PI"`. See the example `bessel_cosine_groundstate.xmds` for a demonstration.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="r" lattice="128" domain="(0, 3)" transform="bessel" volume_prefactor=
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "spherical-bessel" transform

When working in spherical coordinates, it is often useful to use the spherical Bessel functions $j_l(x) = \sqrt{\frac{\pi}{2x}} J_{l+\frac{1}{2}}(x)$ as a basis. These are eigenfunctions of the radial component of Laplace's equation in spherical coordinates:

$$\nabla^2 \left[j_l(kr)\, Y_l^m(\theta,\phi) \right] = \left[\frac{\partial^2}{\partial r^2} + \frac{2}{r}\frac{\partial}{\partial r} - \frac{l(l+1)}{r^2} \right] j_l(kr)\, Y_l^m(\theta,\phi) = -k^2 j_l(kr)\, Y_l^m(\theta,\phi)$$

Just as the Bessel basis above, the transformed dimensions are prefixed with a 'k', and it is possible (and usually wise) to use the `spectral_lattice` attribute to specify a different lattice size in the transformed space. Also, the spacing of these lattices are again chosen in a non-uniform manner to Gaussian quadrature methods for spectrally accurate transforms. Finally, the `order` attribute can be used to specify the order l of the spherical Bessel functions used.

If we denote the transformation to and from this basis by $\mathcal{SH}$, then we can write the useful property:

$$\frac{\partial^2 f}{\partial r^2} + \frac{2}{r}\frac{\partial f}{\partial r} - \frac{l(l+1)}{r^2} = \mathcal{SH}_l^{-1}\left[(-k^2) F_l(k) \right](r)$$

Spherical Bessel transforms allow easy calculation of the Laplacian of fields with spherical symmetry. Applying the operator `L = -kr*kr` in Spherical Bessel space is therefore equivalent to applying the operator

$$\mathcal{L} = \left(\frac{\partial^2}{\partial r^2} + \frac{2}{r}\frac{\partial}{\partial r} - \frac{l(l+1)}{r^2} \right)$$

in coordinate space.

In non-Euclidean co-ordinates, integrals have non-unit volume elements. For example, in spherical co-ordinates with a radial co-ordinate 'r', integrals over this dimension have a volume element $r^2 dr$. When performing integrals along a dimension specified by the "spherical-bessel" transform, the factor of the square of the radius is included implicitly. If you are using a geometry with some symmetry, it is common to have prefactors in your integration. For example, for a three-dimensional volume in spherical symmetry, all integrals would have a volume element of $4\pi r^2 dr$. This extra factor of 4π can be included for all integrals by specifying the attribute `volume_prefactor="4*M_PI"`. This is demonstrated in the example bessel_transform.xmds.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="r" lattice="128" domain="(0, 3)" transform="spherical-bessel" volume_
        </transverse_dimensions>
    </geometry>
</simulation>
```

The "hermite-gauss" transform

The "hermite-gauss" transform allows transformations to and from the basis of Hermite functions $\psi_n(x)$:

$$\psi_n(x) = \left(2^n n! \sigma \sqrt{\pi}\right)^{-1/2} e^{-x^2/2\sigma^2} H_n(\sigma x)$$

where the functions $H_n(x)$ are the Hermite polynomials:

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n}\left(e^{-x^2}\right)$$

which are eigenfunctions of the Schroedinger equation for a harmonic oscillator:

$$-\frac{\hbar^2}{2m}\frac{\partial^2 \psi_n}{\partial x^2} + \frac{1}{2} m\omega^2 x^2 \psi_n(x) = \hbar\omega\left(n + \frac{1}{2}\right)\psi_n(x),$$

with $\sigma = \sqrt{\frac{\hbar}{m\omega}}$.

This transform is different to the others in that it requires a `length_scale` attribute rather than a `domain` attribute, as the range of the lattice will depend on the number of basis functions used. The `length_scale` attribute defines the scale of the domain as the standard deviation σ of the lowest order Hermite function $\psi_0(x)$:

$$\psi_0(x) = (\sigma^2\pi)^{-1/4} e^{-x^2/2\sigma^2}$$

When a dimension uses the "hermite-gauss" transform, then the variable indexing the basis functions is defined as the name of the dimension prefixed with an "n". For example, when referencing the basis function indices for the dimensions "x", "y", "z" and "tau", use the variable "nx", "ny", "nz" and "ntau".

Applying the operator `L = nx + 0.5` in Hermite space is therefore equivalent to applying the operator

$$\mathcal{L} = \left(-\frac{\sigma^2}{2}\frac{\partial^2}{\partial x^2} + \frac{1}{2\sigma^2}x^2\right)$$

in coordinate space.

The Hermite-Gauss transform permits one to work in energy-space for the harmonic oscillator. The normal Fourier transform of "hermite-gauss" dimensions can also be referenced using the dimension name prefixed with a "k". See the examples `hermitegauss_transform.xmds` and `hermitegauss_groundstate.xmds` for examples.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="r" lattice="128" length_scale="1.0" transform="hermite-gauss" />
        </transverse_dimensions>
    </geometry>
</simulation>
```

5.4.8 Vector Element

Vectors are arrays of data, defined over any subset of the transverse dimensions defined in your *Geometry Element*. These dimensions are listed in the attribute `dimensions`, which can be an empty string if you wish the vector to not be defined on any dimensions. If you do not include a `dimensions` attribute then the vector defaults to being a function of all transverse dimensions, not including any aliases. Vectors are used to store static or dynamic variables, but you do not have to specify their purpose when they are defined. They can then be referenced and/or changed by sequence elements, as described below.

Each `<vector>` element has a unique name, defined by a `name` attribute. It is either complex-valued (the default) or real-valued, which can be specified using the `type="real"` attribute. A vector contains a list of variables, each defined by name in the `<components>` element. The name of each component is the name used to reference it later in the simulation.

Vectors are initialised at the beginning of a simulation, either from code or from an input file. The basis choice for this initialisation defaults to the normal space as defined in the `<geometry>` element, but any transverse dimension can be initialised in their transform basis by specifying them in an `initial_basis` attribute. The `initial_basis` attribute lists dimensions either by their name as defined by the `<geometry>` element, or by their transformed name. For example, to initialise a two-dimensional vector defined with `dimensions="x y"` in Fourier space for the y-dimension, we would include the attribute `initial_basis="x ky"`, or just `initial_basis="ky"`. When initialising the vector within the XMDS script, the appropriate code is placed in a 'CDATA' block inside an `<initialisation>` element. This code is in standard C-syntax, and should reference the components of the vector by name. XMDS defines a few useful *shorthand macros* for this C-code. If you wish to initialise all the components of the vector as zeros, then it suffices simply to add the attribute `kind="zero"` or to omit the `<initialisation>` element entirely. While the default XMDS behaviour is to reference all variables locally, any vector can be referenced non-locally. The notation for referencing the value of a vector 'phi' with a dimension 'j' at a value of 'j=jk' is `phi(j => jk)`. Multiple non-local dimensions are addressed by adding the references in a list, e.g. `phi(j => jk, x => y)`. See `2DMultistateSE.xmds` for an example.

Dimensions can only be accessed non-locally if one of the following conditions is true:

- The dimension is an `integer` dimension,
- The dimension is accessed with an *alias* of that dimension. For example, `phi(x => y)` if the dimension `x` has `y` as an alias, or vice-versa.
- The dimension is a Fourier transform dimension (`dft`), used in the spectral basis (i.e. `kx` for an `x` dimension) and it is accessed with the negative of that dimension. For example `phi(kx => -kx)`.
- The dimension is uniformly spaced (i.e. corresponds to the spatial basis of a dimension with a transform of `dft`, `dct`, `dst` or `none`), and it is accessed with the lower limit of that dimension. For example, `phi(x => -1.2)` for a dimension with a domain of `(-1.2, 1.2)`. Note that the dimension must be accessed with the exact characters used in the definition of the domain. For the previous example `phi(x => -1.20)` does not satisfy this condition.

- **Advanced behaviour**: The value of a variable at an arbitrary point can be accessed via the integer index for that dimension. For example `phi(x_index => 3)` accesses the value of `phi` at the grid point with index 3. As `x_index` is zero-based, this will be the *fourth* grid point. It is highly recommended that the *diagnostics* feature be used when writing simulations using this feature. Once the simulation has been tested, `<diagnostics>` can be turned off for data-taking runs.

Note that a dimension cannot be accessed non-locally in `distributed-mpi` simulations if the simulation is distributed across that dimension. If you wish to initialise from a file, then you can choose to initialise from an hdf5 file using `kind="hdf5"` in the `<initialisation>` element, and then supply the name of the input file with the `filename` element. This is a standard data format which can be generated from XMDS, or from another program. An example for generating a file in another program for input into XMDS is detailed in the Advanced topic: *Importing data*.

When initialising from a file, the default is to require the lattice of the transverse dimensions to exactly match the lattice defined by XMDS. There is an option to import data defined on a subset or superset of the lattice points. Obviously, the dimensionality of the imported field still has to be correct. This option is activated by defining the attribute `geometry_matching_mode="loose"`. The default option is defined as `geometry_matching_mode="strict"`. A requirement of the initialisation geometry is that the lattice points of the input file are spaced identically to those of the simulation grid. This allows expanding or contracting a domain between simulations. If used in Fourier space, this feature can be used for coarsening or refining a simulation grid. See *'Loose' geometry_matching_mode* for details.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(-1, 1)" />
        </transverse_dimensions>
    </geometry>

    <!-- A one-dimensional vector with dimension 'x' -->
    <vector name="wavefunction" initial_basis="x" type="complex">
        <components> phi </components>
        <initialisation>
            <![CDATA[
                // 'cis(x)' is cos(x) + i * sin(x)
                phi = exp(-0.5 * x * x) * cis(40 * x);
            ]]>
        </initialisation>
    </vector>

    <!-- A zero-dimensional real vector with components u and v -->
    <vector name="zero_dim" dimensions="" type="real">
        <components>
            u v
        </components>
        <initialisation kind="hdf5">
            <filename>data.h5</filename>
        </initialisation>
    </vector>
</simulation>
```

The dependencies element

Often a vector, computed vector, filter, integration operator or output group will reference the values in one or more other vectors, computed vectors or noise vectors. These dependencies are defined via a `<dependencies>` element, which lists the names of the vectors. The components of those vectors will then be available for use in the 'CDATA' block, and can be referenced by their name.

For a vector, the basis of the dependent vectors, and therefore the basis of the dimensions available in the

'CDATA' block, are defined by the `initial_basis` of the vector. For a `<computed_vector>`, `<filter>` `<integration_vector>`, or moment group vector, the basis of the dependencies can be specified by a `basis` attribute in the `<dependencies>` element. For example, `basis="x ny kz"`.

Any transverse dimensions that appear in the `<dependencies>` element that do not appear in the `dimensions` attribute of the vector are integrated out. For integer dimensions, this is simply an implicit sum over the dimension. For real-valued dimensions, this is an implicit integral over the range of that dimension.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(-1, 1)" />
            <dimension name="y" lattice="10" domain="(-3, 2)" transform="dct" />
        </transverse_dimensions>
    </geometry>

    <!-- A one-dimensional vector with dimension 'x' -->
    <vector name="wavefunction" dimensions="x" initial_basis="x" type="complex">
        <components> phi </components>
        <initialisation>
            <!--
                The initialisation of the vector 'wavefunction' depends on information
                in the 'two_dim' vector.  The vector two_dim is DCT-transformed into the
                (x, ky) basis, and the ky dimension is implicitly integrated over in the
                following initialisation code
              -->
            <dependencies basis="x ky">two_dim</dependencies>
            <![CDATA[
                // 'cis(x)' is cos(x) + i * sin(x)
                phi = exp(-0.5 * x * x + v) * cis(u * x);
            ]]>
        </initialisation>
    </vector>

    <!-- A two-dimensional real vector with components u and v -->
    <vector name="two_dim" type="real">
        <components>
            u v
        </components>
        <initialisation kind="hdf5">
            <filename>data.h5</filename>
        </initialisation>
    </vector>
</simulation>
```

5.4.9 Computed Vector Element

Computed vectors are arrays of data much like normal `<vector>` elements, but they are always calculated as they are referenced, so they cannot be initialised from file. It is defined with a `<computed_vector>` element, which has a `name` attribute, optional `dimensions` and `type` attributes, and a `<components>` element, just like a `<vector>` element. Instead of an *<initialisation>* element, it has an `<evaluation>` element that serves the same purpose. The `<evaluation>` element contains a `<dependencies>` element (see `above<Dependencies>`), and a 'CDATA' block containing the code that defines it.

As it is not being stored, a `<computed_vector>` does not have or require an `initial_basis` attribute, as it will be transformed into an appropriate basis for the element that references it. The basis for its evaluation will be determined entirely by the `basis` attribute of the `<dependencies>` element.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(-1, 1)" />
        </transverse_dimensions>
    </geometry>

    <!-- A one-dimensional vector with dimension 'x' -->
    <vector name="wavefunction" type="complex">
        <components> phi </components>
        <initialisation>
            <![CDATA[
                // 'cis(x)' is cos(x) + i * sin(x)
                phi = exp(-0.5 * x * x) * cis(40 * x);
            ]]>
        </initialisation>
    </vector>

    <!-- A zero-dimensional real computed vector with components Ncalc -->
    <computed_vector name="zero_dim" dimensions="" type="real">
        <components>
            Ncalc
        </components>
        <evaluation>
            <dependencies>wavefunction</dependencies>
            <![CDATA[
                // Implicitly integrating over the dimension 'x'
                Ncalc = mod2(phi);
            ]]>
        </evaluation>
    </computed_vector>
</simulation>
```

5.4.10 Noise Vector Element

Noise vectors are used like computed vectors, but when they are evaluated they generate arrays of random numbers of various kinds. They do not depend on other vectors, and are not initialised by code. They are defined by a `<noise_vector>` element, which has a `name` attribute, and optional `dimensions`, `initial_basis` and `type` attributes, which work identically as for normal vectors.

The choice of pseudo-random number generator (RNG) can be specified with the `method` attribute, which has options "posix" (the default), "mkl", "solirte" and "dsfmt". It is only possible to use any particular method if that library is available. Although "posix" is the default, it is also the slowest, and produces the lowest quality random numbers (although this is typically not a problem). "mkl" refers to the Intel Math Kernel Library, and is only available if installed. "solirte" and "dsfmt" are fast, hardware-accelerated random number sources that should work on most systems. "mkl", "solirte" and "dsfmt" have comparable performance.

The random number generators can be provided with a seed using the `seed` attribute, which should typically consist of a list of three integers. All RNGs require positive integers as seeds. It is possible to use the *<validation kind="run-time"/>* feature to use passed variables as seeds. It is advantageous to use fixed seeds rather than timer-based seeds, as the *<error_check>* element can test for strong convergence if the same seeds are used for both integrations. If the `seed` attribute is not specified, then seeds will be generated at the time the simulation is run. Different executions of the same simulation will therefore give different results. However, results can be reproduced by examining the `.xsil` file produced by the simulation which contains the generated seeds. If these seeds are used for the `seed` attribute, the same results can be reproduced. Unless you need to reproduce particular results, it is unnecessary to specify the `seed` attribute.

The different types of noise vectors are defined by a mandatory `kind` attribute, which must take the value of 'gauss', 'gaussian', 'wiener', 'poissonian','jump' or 'uniform'.

Example syntax:

```
<simulation xmds-version="2">
    <geometry>
        <propagation_dimension> t </propagation_dimension>
        <transverse_dimensions>
            <dimension name="x" lattice="128" domain="(-1, 1)" />
        </transverse_dimensions>
    </geometry>

    <!--
        A one-dimensional complex wiener noise vector.
        This noise is appropriate for using in the complex
        random-walk equation of motion:
            dz_dt = eta;
    -->
    <noise_vector name="noise" kind="wiener">
        <components>
            eta
        </components>
    </vector>
</simulation>
```

Uniform noise

Uniform noises defined over any transverse dimensions are simply uniformly distributed random numbers between zero and one. This noise is an example of a "static" noise, i.e. one suitable for initial conditions of a field. If it were included in the equations of motion for a field, then the effect of the noise would depend on the lattice spacing of the propagation dimension. XMDS therefore does not allow this noise type to be used in integration elements.

Example syntax:

```
<simulation xmds-version="2">
    <noise_vector name="drivingNoise" dimensions="x" kind="uniform" type="complex" method="dsfmt"
      <components>Eta</components>
    </noise_vector>
</simulation>
```

Gaussian noise

Noise generated with the "gaussian" method is gaussian distributed with zero mean. For a real-valued noise vector, the variance at each point is the inverse of the volume element of the transverse dimensions in the vector. This volume element for a single transverse dimension is that used to perform integrals over that dimension. For example, it would include a factor of r^2 for a dimension "r" defined with a `spherical-bessel` transform. It can be non-uniform for dimensions based on non-Fourier transforms, and will include the product of the `volume_prefactor` attribute as specified in the *Geometry* element. The volume element for an integer-type dimension is unity (i.e. where the integral is just an unweighted sum). The volume element for a `noise_vector` with multiple dimensions is simply the product of the volume elements of the individual dimensions.

This lattice-dependent variance is typical in most applications of partial differential equations with stochastic initial conditions, as the physical quantity is the variance of the field over some finite volume, which does not change if the variance at each lattice site varies as described above.

For complex-valued noise vector, the real and imaginary parts of the noise are independent, and each have half the variance of a real-valued noise. This means that the modulus squared of a complex-valued noise vector has the same variance as a real-valued noise vector at each point.

Gaussian noise vectors are an example of a "static" noise, i.e. one suitable for initial conditions of a field. If they were included in the equations of motion for a field, then the effect of the noise would depend on the lattice spacing of the propagation dimension. XMDS therefore does not allow this noise type to be used in integration elements.

Example syntax:

```
<simulation xmds-version="2">
    <noise_vector name="initialNoise" dimensions="x" kind="gauss" type="real" method="posix" seed=
      <components>fuzz</components>
    </noise_vector>
</simulation>
```

Wiener noise

Noise generated with the "wiener" method is gaussian distributed with zero mean and the same variance as the static "gaussian" noise defined above, multiplied by a factor of the lattice step in the propagation dimension. This means that these noise vectors can be used to define Wiener noises for standard stochastic ordinary or partial differential equations. Most integrators in XMDS effectively interpret these noises as Stratonovich increments.

As a dynamic noise, a Wiener process is not well-defined except in an `integrate` element.

Example syntax:

```
<simulation xmds-version="2">
    <noise_vector name="diffusion" dimensions="x" kind="wiener" type="real" method="solirte" seed=
      <components>dW</components>
    </noise_vector>
</simulation>
```

Poissonian noise

A noise vector using the "poissonian" method generates a random variable from a Poissonian distribution. While the the Poisson distribution is integer-valued, the variable will be cast as a real number. The rate of the Poissonian distribution is defined by the `mean` or `mean-density` attributes. These are are synonyms, and must be defined as positive real numbers. For Poissonian noises defined over real-valued transverse dimensions, the rate is given by the product of this `mean-density` attribute and the volume element at that point, taking into account all transverse dimensions, including their `volume_prefactor` attributes. The result is that the integral over each volume in space is a sample from a Poissonian distribution of that rate.

Poissonian noise vectors are an example of a "static" noise, i.e. one suitable for initial conditions of a field. If they were included in the equations of motion for a field, then the effect of the noise would depend on the lattice spacing of the propagation dimension. XMDS therefore does not allow this noise type to be used in integration elements.

Example syntax:

```
<simulation xmds-version="2">
    <noise_vector name="initialDistribution" dimensions="x" kind="poissonian" type="real" mean-de
      <components>Pdist</components>
    </noise_vector>
</simulation>
```

Jump noise

A noise vector using the "jump" method is the dynamic version of the poissonian noise method, and must have the `mean-rate` attribute specified as a positive real number. The variable at each point is chosen from a Poissonian distribution with a mean equal to the product of three variables: the `mean-rate` attribute; the volume of the element as defined by its transverse dimensions (including their `volume_prefactor` attributes); and the step size in the propagation dimension. Normally defined in the limit where the noise value is zero almost always, with a few occurrences where it is unity, and none of any higher value, this type of noise is commonly used in differential equations with a Poissonian jump process.

It is common to wish to vary the mean rate of a jump process, which means that the `mean-rate` attribute must be a variable or a piece of code. These cannot be verified to be a positive real number at compile time, so they must be used with the *<validation>* feature with either the `kind="none"` or `kind="run-time"` attributes.

As a dynamic noise, a jump process is not well-defined except in an `integrate` element.

Example syntax:

```
<simulation xmds-version="2">
    <noise_vector name="initialDistribution" dimensions="" kind="jump" type="real" mean-rate="2.7
      <components>dN</components>
    </noise_vector>
</simulation>
```

5.4.11 Sequence Element

All processing of vectors happens in sequence elements. Each simulation must have exactly one main sequence element, but it can then contain any number of nested sequence elements. A sequence element can contain any number of `<sequence>`, *<filter>*, *<integrate>* and/or *<breakpoint>* elements, which are executed in the order they are written. A sequence can be repeated a number of times by using the `cycles` attribute. For example, `<sequence cycles="10">` will execute the elements in that sequence 10 times.

Example syntax:

```
<simulation xmds-version="2">
    <sequence cycles="2">
        <sequence>  ... </sequence>
        <filter> ... </filter>
        <integrate> ...</integrate>
    </sequence>
</simulation>
```

5.4.12 Filter element

A `<filter>` element can be placed inside a `<sequence>` element or an *<integrate>* element. It contains a 'CDATA' block and an optional *<dependencies>* element, which may give access to variables in other `<vector>`, `<computed_vector>` or `<noise_vector>` elements. The code inside the 'CDATA' block is executed over the combined tensor product space of the dependencies, or simply once if there is no dependencies element. This element therefore allows arbitrary execution of C-code.

Sometimes it is desirable to apply a filter conditionally. The most efficient way of doing this is to call the function from the piece of code that contains the conditional statement (likely another `<filter>` element) rather than embed the conditional function in the filter itself, as the latter method can involve the conditional statement being evaluated multiple times over the transverse dimensions. For this reason, it is possible to give a filter a `name` attribute, and the filter can thenceforth be called in CDATA blocks by that name. For example: `<filter name="filterName">` allows the function to be called using the C-function `filterName()`.

One of the common uses of a filter element is to apply discontinuous changes to the vectors and variables of the simulation.

Example syntax:

```
<sequence>
    <filter>
      <![CDATA[
        printf("Hello world from the first filter segment!  This filter rather wastefully calls t
        fname();
      ]]>
    </filter>

    <filter name="fname">
       <dependencies>normalisation wavefunction</dependencies>
```

```
      <![CDATA[
        phi *= sqrt(Nparticles/Ncalc);
      ]]>
    </filter>
</sequence>
```

5.4.13 Integrate element

The <integrate> element is at the heart of most XMDS simulations. It is used to integrate a set of (potentially stochastic) first-order differential equations for one or more of the vectors defined using the <vector> element along the propagation dimension. At the beginning of the simulation, the value of the propagation dimension is set to zero, and the vectors are initialised as defined in the *<vector>* element. As successive sequence elements change these variables, each integrate element simply integrates onward from the current values.

The length of the integration is defined by the interval attribute, which must be a positive real number. An <integrate> element must have an algorithm attribute defined, which defines the integration method. Current methods include *SI*, *SIC*, *RK4*, *RK9*, *ARK45*, and *ARK89*. Fixed step algorithms require a steps attribute, which must be a positive integer that defines the number of (evenly spaced) integration steps. Adaptive stepsize algorithms require a tolerance attribute that must be a positive real number much smaller than one, which defines the allowable relative error per integration step. If the steps attribute is specified for an adaptive stepsize algorithm, then it is used to generate the initial stepsize estimate. The optional <samples> element is used to track the evolution of one or more vectors or variables during an integration. This element must contain a non-negative integer for each *<sampling_group>* element defined in the simulation's *<output>* element. The list of integers then defines the number of times that the moments defined in those groups will be sampled. For a fixed step algorithm, each non-zero number of samples must be a factor of the total number of steps.

The vectors to be integrated and the form of the differential equations are defined in the *<operators>* element (or elements). Filters to be applied each step can be defined with optional *<filters>* elements.

Computed vectors can be defined with the <computed_vector> element. These act exactly like a globally defined *Computed Vector Element*, but are only available within the single <integrate> element.

Example syntax:

```
<integrate algorithm="ARK89" interval="1e-4" steps="10000" tolerance="1e-8">
  <samples>20</samples>
  <filters>
    <filter>
      <dependencies>wavefunction normalisation</dependencies>
      <![CDATA[
        phi *= sqrt(Nparticles/Ncalc);   // Correct normalisation of the wavefunction
      ]]>
    </filter>
  </filters>
  <operators>
    <operator kind="ip" constant="yes">
      <operator_names>T</operator_names>
      <![CDATA[
        T = -0.5*hbar/M*ky*ky;
      ]]>
    </operator>
    <dependencies>potential</dependencies>
    <![CDATA[
      dphi_dt = T[phi] - (V1 + Uint/hbar*mod2(phi))*phi;
    ]]>
    <integration_vectors>wavefunction</integration_vectors>
  </operators>
</integrate>
```

Operators and operator elements

An *<integrate>* element must contain one or more `<operators>` elements, which define both which vectors are to be integrated, and their derivative in the propagation dimension. When all vectors to be integrated have the same dimensionality, they can all be defined within a single `<operators>` element, and when vectors with different dimension are to be integrated, each set of vectors with the same dimensionality should be placed in separate `<operators>` elements. Within each `<operators>` element, the vectors that are to be integrated are listed by name in the `<integration_vectors>` element, and the differential equations are written in a 'CDATA' block. The derivative of each component of the integration vectors must be defined along the propagation dimension. For example, if the integration vectors have components 'phi' and 'beta', and the propagation dimension is labelled 'tau', then the 'CDATA' block must define the variables 'dphi_dtau' and 'dbeta_dtau'. These derivatives can be any function of the available variables, including any components from other vectors, computed vectors or noise vectors that are listed in the optional *<dependencies>* element. These dependent vectors must be defined on a subset of the dimensions of the integration vectors.

When noise vectors are referenced, equations with Wiener noises should be written as though the equations are in differential form, as described in the worked examples *Kubo Oscillator* and *Fibre Noise*. Jump-based Poisson noises will also be written in an equivalent form, as modelled by the example 'photodetector.xmds'.

By default, the name of each component references the local value of the vector, but *nonlocal variables* can be accessed using the standard syntax. However, typically the most common (and most efficient) method of referencing nonlocal variables is to reference variables that are local in the *transformed space* for a given transverse dimension. This is done using `<operator>` elements. There are three kinds of `<operator>` elements. The first is denoted with a `kind="functions"` attribute, and contains a 'CDATA' block that will be executed in the order that it is defined. This is useful when you wish to calculate functions that do not depend on the transverse dimensions. Defining these along with the main equations of motion causes them to be recalculated separately for each point. The second kind of `<operator>` element is used to define an operation in a transformed space. This is often an efficient method of calculating common nonlocal terms such as derivatives. The third kind is used to define integration of one or more vectors along a transverse dimension.

Example syntax:

```
<operator kind="functions">
  <![CDATA[
  f = cos(t);
  ]]>
</operator>
```

The second kind of operator element defines a list of operators in an `<operator_names>` element. The basis of these operators defaults to the transform space unless a different basis is specified using the `basis` attribute. These operators must then be defined in a 'CDATA' block, using any *dependencies* as normal. If the operators constant across the integration, then the attribute `constant="yes"` should be set, otherwise the `constant="no"` attribute ensures that the operator is recalculated each step. The operators defined in these elements can then be used in the 'CDATA' block that defines the equations of motion. The application of operator 'L' to vector 'psi' is denoted `L[psi]`. Operators can be applied to functions of vectors using the same notation, such as `L[psi*psi]`. Aside from the example above, many examples can be found in the examples folder, and the *Worked Examples* section of the documentation.

Operators of this second kind have the `kind="IP"` or `kind="EX"` attribute, standing for 'interaction picture' and 'explicit' operators respectively. Explicit operators can be used in all situations, and simply construct and calculate a new vector of the form in the square brackets. IP operators use less memory and can improve speed by allowing larger timesteps, but have two important restrictions. **Use of IP operators without understanding these restrictions can lead to incorrect code**. The first restriction is that IP operators can only be applied to named components of one of the integration vectors, and not functions of those components. The second restriction is that the equations of motion must be written such that the term with the operator is not multiplied by any quantity or used inside a function. (For those interested, the reason for this is that the IP algorithm applies the operator separately to the rest of the evolution, and therefore the actual text of the `L[psi]` term is replaced by the numeral zero.) If you must break either of those rules, then you need to use the EX algorithm.

Example syntax:

```
<operator kind="ex" constant="yes">
  <operator_names>T</operator_names>
  <![CDATA[
    T = -0.5*hbar/M*ky*ky;
  ]]>
</operator>
```

The third kind of operator element is used to define an integration along a transverse dimension. This kind of evolution is called "cross-propagation", and is described briefly in the examples 'tla.xmds', 'tla_sic.xmds' and 'sine_cross.xmds'. This class of equations have a subset of vectors that have an initial condition on one side of a transverse dimension, and a differential equation defined in that dimension, and as such, this kind of operator element has much of the structure of an entire *<integrate>* element.

An operator element with the `kind="cross_propagation"` attribute must specify the transverse dimension along which the integration would proceed with the `propagation_dimension` attribute. It must also specify its own *<integration_vectors>* element, its own `<operators>` elements (of the second kind), and may define an optional *<dependencies>* element. The algorithm to be used for the transverse integration is specified by the `algorithm` attribute, with options being `algorithm="SI"` and `algorithm="RK4"`. The derivatives in the cross propagation direction are defined in a 'CDATA' block, just as for a normal `<integrate>` element. The boundary conditions are specified by a `<boundary_conditions>` element, which requires the `kind="left"` or `kind="right"` attribute to specify on which side of the grid that the boundary conditions are specified. The boundary conditions for the `<integration_vectors>` are then specified in a 'CDATA' block, which may refer to vectors in an optional *<dependencies>* element that can be contained in the `<boundary_conditions>` element.

Example syntax:

```
<operator kind="cross_propagation" algorithm="RK4" propagation_dimension="t">
  <integration_vectors>cross</integration_vectors>
  <dependencies>constants</dependencies>
  <boundary_condition kind="left">
    <![CDATA[
      v = 1.0;
      w = 1.0;
    ]]>
  </boundary_condition>

  <operator kind="ip" constant="yes">
    <operator_names>L</operator_names>
    <![CDATA[
      L = i;
    ]]>
  </operator>

  <![CDATA[
    dv_dt = i*v;
    dw_dt = L[w];
  ]]>
</operator>
```

Algorithms

The stability, efficiency and even convergence of a numerical integration can depend on the method. Due to the varying properties of different sets of equations, it is impossible to define the best method for all equations, so XMDS provides an option to use different algorithms. These include fixed step algorithms, which divide the integration region into equal steps, and adaptive stepsize algorithms, which attempt to estimate the error in the simulation in order to choose an appropriate size for the next step. As a first guess, a good method for a deterministic integration would be *ARK89*, and a good guess for a stochastic method would be the *SI and SIC algorithms*.

For the purposes of the descriptions below, we will assume that we are considering the following set of coupled

differential equations for the vector of variables $\mathbf{x}(t)$:

$$\frac{dx_j}{dt} = f_j(\mathbf{x}(t), t)$$

SI and SIC algorithms

The SI algorithm is a semi-implicit fixed-step algorithm that finds the increment of the vector by solving

$$x_j(t+\Delta t) = x_j(t) + f_j\left(\mathbf{x}(t+\frac{\Delta t}{2}), t+\frac{\Delta t}{2}\right)\ \Delta t$$

using a simple iteration to find the values of the vector at the midpoint of the step self-consistently. The number of iterations can be set using the `iterations` attribute, and it defaults to `iterations="3"`. The choice of `iterations="1"` is therefore fully equivalent to the Euler algorithm, where

$$x_j(t+\Delta t) = x_j(t) + f_j\left(\mathbf{x}(t), t\right)\ \Delta t.$$

The Euler algorithm is the only safe algorithm for direct integration of *jump-based Poisson processes*. Efficient numerical solution of those types of equations is best done via a process of triggered filters, which will be described in the *Advanced Topics* section.

When SI integration is used in conjunction with SI cross-propagation, a slight variant of the SI algorithm can be employed where the integration in both directions is contained within the iteration process. This is activated by using `algorithm="SIC"` rather than `algorithm="SI"`.

The SI algorithm is correct to second order in the step-size for deterministic equations, and first order in the step-size for Stratonovich stochastic equations with Wiener noises. This makes it the highest order stochastic algorithm in XMDS, although there are many sets of equations that integrate more efficiently with lower order algorithms.

Runge-Kutta algorithms

Runge-Kutta algorithms are the workhorse of numerical integration, and XMDS employs two fixed step versions: `algorithm="RK4"`, which is correct to fourth-order in the step size, and `algorithm="RK9"`, which is correct to ninth order in the step size. It must be strongly noted that a higher order of convergence does not automatically mean a superior algorithm. RK9 requires several times the memory of the RK4 algorithm, and each step requires significantly more computation.

All Runge-Kutta algorithms are convergent for Stratonovich stochastic equations at the order of the square root of the step-size. This 'half-order' convergence may seem very weak, but for some classes of stochastic equation this improves up to one half of the deterministic order of convergence. Also, the convergence of some stochastic equations is limited by the 'deterministic part', which can be improved dramatically by using a higher order Runge-Kutta method.

Adaptive Runge-Kutta algorithms

Fixed step integrators can encounter two issues. First, as the equations or parameters of a simulation are changed, the minimum number of steps required to integrate it may change. This means that the convergence must be re-tested multiple times for each set of parameters, as overestimating the number of steps required to perform an integration to a specified error tolerance can be very inefficient. Second, even if the minimum acceptable number of steps required is known for a given simulation, it may be that there are regions of integration that are of wildly varying difficulty. For a fixed step integrator, this means that the step-size must be small enough to handle the most difficult region, and is therefore inefficiently small for the easier regions. Adaptive step-size algorithms get around this problem by testing the convergence during the integration, and adjusting the step-size until it reaches some target tolerance.

XMDS employs two adaptive step-size algorithms based on 'embedded Runge-Kutta' methods. These are Runge-Kutta methods that can output multiple variables that have different convergence. The difference between the higher-order and the lower-order solutions gives an estimate of the error in each step, which can then be used to

estimate an appropriate size for the next step. We use `algorthim="ARK45"`, which contains fourth and fifth order solutions, and `algorthim=ARK89`, which contains eighth and ninth order solutions. Each algorithm converges with the order of the lowest order solution (fourth and eighth order respectively). The overheads involved in estimating the error and step-size make the adaptive algorithms slower than fixed step integration using the same step-size, but overall there is typically a significant performance gain from being able to avoid doing this optimisation manually.

All adaptive stepsize algorithms require a `tolerance` attribute, which must be a positive real number that defines the allowable error per step. It is also possible to specify a `max_iterations` attribute, which is a positive integer that stops the integrator from trying too many times to find an acceptable stepsize. The integrator will abort with an error if the number of attempts for a single step exceeds the maximum specified with this attribute.

As all Runge-Kutta solutions have equal order of convergence for stochastic equations, *if the step-size is limited by the stochastic term then the step-size estimation is entirely unreliable*. Adaptive Runge-Kutta algorithms are therefore not appropriate for stochastic equations.

Filters element

Filter elements are used inside *sequence elements* to execute arbitrary code, or make discontinuous changes in the vectors. Sometimes it is desirable to perform a filter element at the beginning or end of each step in an integration. This can be done by placing `<filter>` elements in a `<filters>` element within the `<integrate>` element. The `<filters>` element specifies whether the filters are to be executed at the end of each step or the beginning of each step with the `where="step end"` and `where="step start"` attributes respectively. Each filter is then executed in the order found in the `<filters>` element.

Example syntax:

```
<integrate algorithm="ARK45" interval="100000.0" steps="10000000" tolerance="1e-8">
  <samples>5000 100</samples>
  <filters where="step end">
    <filter>
        <dependencies>vector1 vector2</dependencies>
        <![CDATA[
            x = 1;
            y *= ynorm;
            ]]>
    </filter>
  </filters>

  <operators>
    <integration_vectors>vector1</integration_vectors>
    <![CDATA[
    dx_dt = alpha;
    dy_dt = beta*y;
    ]]>
  </operators>
</integrate>
```

5.4.14 Breakpoint element

The `<breakpoint>` element is used to output the full state of one or more vectors. Unlike sampled output, it executes immediately rather than at the end of a program, and can therefore be used to examine the current state of an ongoing simulation. The vectors to be output are defined via a *<dependencies>* element, and the basis is chosen by the `basis` attribute supplied to that `<dependencies>` element, as usual. A single `<breakpoint>` element must only contain vectors of equal dimension. The data format is specified by the `format` attribute, with current options being "ascii", "binary" and the recommended: "hdf5". The filename for the output can be specified by a `filename` attribute, in which case the same filename will be used each time the element is executed. If the `filename` attribute is not specified, then the first output will default to "1.xsil", and subsequent executions of the same breakpoint will increment the number by one.

Example syntax:

```
<breakpoint filename="groundstate_break.xsil" format="hdf5">
  <dependencies basis="ky">wavefunction</dependencies>
</breakpoint>
```

5.4.15 Output element

The <output> element describes the output of the program. It is often inefficient to output the complete state of all vectors at all times during a large simulation, so the purpose of this function is to define subsets of the information required for output. Each different format of information is described in a different <sampling_group> element inside the output element. The <output> element may contain any number of <sampling_group> elements. The format of the output data can be specified by the optional format attribute, which may take values of "ascii", "binary", and "hdf5" (the default). The filename can be specified with the optional filename element, which otherwise defaults to the simulation name with the '.xsil' suffix.

The <samples> inside <integrate> elements defines a string of integers, with exactly one for each <sampling_group> element. During that integration, the variables described in each <sampling_group> element will be sampled and stored that number of times.

Sampling Group Element

A <sampling_group> element defines a set of variables that we wish to output, typically they are functions of some subset of vectors. The names of the desired variables are listed in a <moments> element, just like the <components> element of a vector. They are defined with a '*CDATA*' block, accessing any components of vectors and computed vectors that are defined in a *<dependencies>* element, also just like a vector. *Computed vectors* and *<operator>* elements can be defined and used in the definition, just like in an *<integrate>* element.

The basis of the output is specified by the basis attribute. This overrides any basis specification in the <dependencies> element. Because we often wish to calculate these vectors on a finer grid than we wish to output, it is possible to specify that the output on a subset of the points defined for any transverse dimension. This is done by adding a number in parentheses after that dimension in the basis string, e.g. basis="x y(32) kz(64)". If the number is zero, then that dimension is integrated out. If that number is one or more, then that dimension will be sampled on a subset of points in that space.

The initial_sample attribute, which must be "yes" or "no", determines whether the moment group will be sampled before any integration occurs.

Example syntax:

```
<output format="hdf5" filename="SimOutput.xsil">
  <sampling_group basis="x y" initial_sample="yes">
    <computed_vector name="filter3" dimensions="" type="complex">
      <components>sparemomentagain</components>
      <evaluation>
        <dependencies basis="kx ky">integrated_u main</dependencies>
        <![CDATA[
          sparemomentagain = mod2(u);
        ]]>
      </evaluation>
    </computed_vector>
    <operator kind="ex" constant="no">
      <operator_names>L</operator_names>
      <![CDATA[
        L = -T*kx*kx/mu;
      ]]>
    </operator>
    <moments>amp ke</moments>
    <dependencies>main filter1</dependencies>
    <![CDATA[
      amp = mod2(u + moment);
```

```
      ke = mod2(L[u]);
    ]]>
  </sampling_group>

  <sampling_group basis="kx(0) ky(64)" initial_sample="yes">
    <moments>Dens_P </moments>
    <dependencies>fields </dependencies>
    <![CDATA[
      Dens_P = mod2(psi);
    ]]>
  </sampling_group>
</output>
```

5.4.16 XMDS-specific C syntax

Sampling complex numbers can be written more efficiently using:

```
<![CDATA[
  _SAMPLE_COMPLEX(W);
]]>
```

which is short for

```
<![CDATA[
  WR = W.Re();
  WI = W.Im();
]]>
```

CHAPTER

SIX

ADVANCED TOPICS

This section has further details on some important topics.

Importing data (importing data into XMDS2, and data formats used in the export)

Convolutions and Fourier transforms (extra information on the Fourier transforms used in XMDS2, and applications to defining convolutions)

Dimension aliases (dimensions which are declared to be identical, useful for correlation functions)

6.1 Importing data

There are many cases where it is advantageous to import previously acquired data into XMDS2. For example, the differential equation you wish to solve may depend on a complicated functional form, which is more easily obtained via an analytical package such as Mathematica or Maple. Furthermore, importing data from another source can be quicker than needlessly performing calculations in XMDS2. In this tutorial, we shall consider an example of importing into XMDS2 a function generated in Mathematica, version 6.0. Note, however, that in order to do this it is required that hdf5 is installed (see http://www.hdfgroup.org/HDF5/).

Suppose we want to import the following function into XMDS2:

$$f(x) = x^2$$

The first step is to create an hdf5 file, from XMDS2, which specifies the dimensions of the grid for the x dimension. Create and save a new XMDS2 file. For the purposes of this tutorial we shall call it "grid_specifier.xmds" with name "grid_specifier". Within this file, enter the following "dummy" vector - which we shall call "gen_dummy" - which depends on the x dimension:

```
<vector type="real" name="gen_dummy" dimensions="x">
  <components>dummy</components>
  <initialisation>
  <![CDATA[
    dummy = x;
      ]]>
  </initialisation>
</vector>
```

What "dummy" is is not actually important. It is only necessary that it is a function of x. However, it is important that the domain and lattice for the x dimension are identical to those in the XMDS2 you plan to import the function into. We output the following xsil file (in hdf5 format) by placing a breakpoint in the sequence block as follows:

```
<sequence>
  <breakpoint filename="grid.xsil" format="hdf5">
      <dependencies>
        gen_dummy
      </dependencies>
  </breakpoint>
```

In terminal, compile the file "grid_specifier.xmds" in XMDS2 and run the c code as usual. This creates two files - "grid.xsil" and "grid.h5". The file "grid.h5" contains the list of points which make up the grids for the x dimensions. This data can now be used to ensure that the function $f(x)$ which we will import into XMDS2 is compatible with the the specified grid in your primary XMDS2 file.

In order to read the "grid.h5" data into Mathematica version 6.0, type the following command into terminal:.. code-block:

```
xsil2graphics2 -e grid.xsil
```

This creates the Mathematica notebook "grid.nb". Open this notebook in Mathematica and evaluate the first set of cells. This has loaded the grid information into Mathematica. For example, suppose you have specified that the x dimension has a lattice of 128 points and a domain of (-32, 32). Then calling "x1" in Mathematica should return the following list:

```
{-32., -31.5, -31., -30.5, -30., -29.5, -29., -28.5, -28., -27.5,
-27., -26.5, -26., -25.5, -25., -24.5, -24., -23.5, -23., -22.5,
-22., -21.5, -21., -20.5, -20., -19.5, -19., -18.5, -18., -17.5,
-17., -16.5, -16., -15.5, -15., -14.5, -14., -13.5, -13., -12.5,
-12., -11.5, -11., -10.5, -10., -9.5, -9., -8.5, -8., -7.5, -7.,
-6.5, -6., -5.5, -5., -4.5, -4., -3.5, -3., -2.5, -2., -1.5, -1.,
-0.5, 0., 0.5, 1., 1.5, 2., 2.5, 3., 3.5, 4., 4.5, 5., 5.5, 6., 6.5,
7., 7.5, 8., 8.5, 9., 9.5, 10., 10.5, 11., 11.5, 12., 12.5, 13.,
13.5, 14., 14.5, 15., 15.5, 16., 16.5, 17., 17.5, 18., 18.5, 19.,
19.5, 20., 20.5, 21., 21.5, 22., 22.5, 23., 23.5, 24., 24.5, 25.,
25.5, 26., 26.5, 27., 27.5, 28., 28.5, 29., 29.5, 30., 30.5, 31.,
31.5}
```

This is, of course, the list of points which define our grid.

We are now in a position to define the function $f(x)$ in Mathematica. Type the following command into a cell in the Mathematica notebook "grid.nb":

```
f[x_]:= x^2
```

At this stage this is an abstract mathematical function as defined in Mathematica. What we need is a list of values for $f(x)$ corresponding to the specified grid points. We will call this list "func". This achieved by simply acting the function on the list of grid points "x1":

```
func := f[x1]
```

For the example grid mentioned above, calling "func" gives the following list:

```
{1024., 992.25, 961., 930.25, 900., 870.25, 841., 812.25, 784.,
756.25, 729., 702.25, 676., 650.25, 625., 600.25, 576., 552.25, 529.,
506.25, 484., 462.25, 441., 420.25, 400., 380.25, 361., 342.25, 324.,
306.25, 289., 272.25, 256., 240.25, 225., 210.25, 196., 182.25, 169.,
156.25, 144., 132.25, 121., 110.25, 100., 90.25, 81., 72.25, 64.,
56.25, 49., 42.25, 36., 30.25, 25., 20.25, 16., 12.25, 9., 6.25, 4.,
2.25, 1., 0.25, 0., 0.25, 1., 2.25, 4., 6.25, 9., 12.25, 16., 20.25,
25., 30.25, 36., 42.25, 49., 56.25, 64., 72.25, 81., 90.25, 100.,
110.25, 121., 132.25, 144., 156.25, 169., 182.25, 196., 210.25, 225.,
240.25, 256., 272.25, 289., 306.25, 324., 342.25, 361., 380.25, 400.,
420.25, 441., 462.25, 484., 506.25, 529., 552.25, 576., 600.25, 625.,
650.25, 676., 702.25, 729., 756.25, 784., 812.25, 841., 870.25, 900.,
930.25, 961., 992.25}
```

The next step is to export the list "func" as an h5 file that XMDS2 can read. This is done by typing the following command into a Mathematica cell:

```
SetDirectory[NotebookDirectory[]];
Export["func.h5", {func, x1}, {"Datasets", { "function_x", "x"}}]
```

In the directory containing the notebook "grid.nb" you should now see the file "func.h5". This file essentially contains the list `{func, x1}`. However, the hdf5 format stores func and x1 as separate entities called "Datasets".

For importation into XMDS2 it is necessary that these datasets are named. This is precisely what the segment `{"Datasets", { "function_x", "x"}}` in the above Mathematica command does. The dataset corresponding to the grid x1 needs to be given the name of the dimension that will be used in XMDS2 - in our case this is "x". It does not matter what the name of the dataset corresponding to the list "func" is; in our case it is "function_x".

The final step is to import the file "func.h5" into your primary XMDS2 file. This data will be stored as a vector called "gen_function_x", in component "function_x".

```
<vector type="real" name="gen_function_x" dimensions="x">
  <components>function_x</components>
  <initialisation kind="hdf5">
    <filename> function_x.h5 </filename>
  </initialisation>
</vector>
```

You're now done. Anytime you want to use $f(x)$ you can simply refer to "function_x" in the vector "gen_function_x".

The situation is slightly more complicated if the function you wish to import depends on more than one dimension. For example, consider

$$g(x, y) = x \sin(y)$$

As for the single dimensional case, we need to export an hdf5 file from XMDS2 which specifies the dimensions of the grid. As in the one dimensional case, this is done by creating a dummy vector which depends on all the relevant dimensions:

```
<vector type="real" name="gen_dummy" dimensions="x y">
  <components>dummy</components>
  <initialisation>
  <![CDATA[
    dummy = x;
      ]]>
  </initialisation>
</vector>
```

and exporting it as shown above.

After importing the grid data into Mathematica, define the multi-dimensional function which you wish to import into XMDS2:

```
g[x_,y_]:= x*Sin[y]
```

We need to create a 2x2 array of data which depends upon the imported lists x1 and y1. This can be done by using the Table function:

```
func := Table[g[x, p], {x, x1}, {p, p1}]
```

This function can be exported as an h5 file,

```
SetDirectory[NotebookDirectory[]];
Export["func.h5", {func, x1, y1}, {"Datasets", { "function_x", "x", "y"}}]
```

and imported into XMDS2 as outlined above.

6.2 Convolutions and Fourier transforms

When evaluating a numerical Fourier transform, XMDS2 doesn't behave as expected. While many simulations have ranges in their spatial coordinate (here assumed to be x) that range from some negative value x_{min} to some positive value x_{max}, the Fourier transform used in XMDS2 treats all spatial coordinates as starting at zero. The

result of this is that a phase factor of the form $e^{-ix_{\min}k}$ is applied to the Fourier space functions after all forward (from real space to Fourier space) Fourier transforms, and its conjugate is applied to the Fourier space functions before all backward (from Fourier space to real space) Fourier transforms.

The standard Fourier transform is

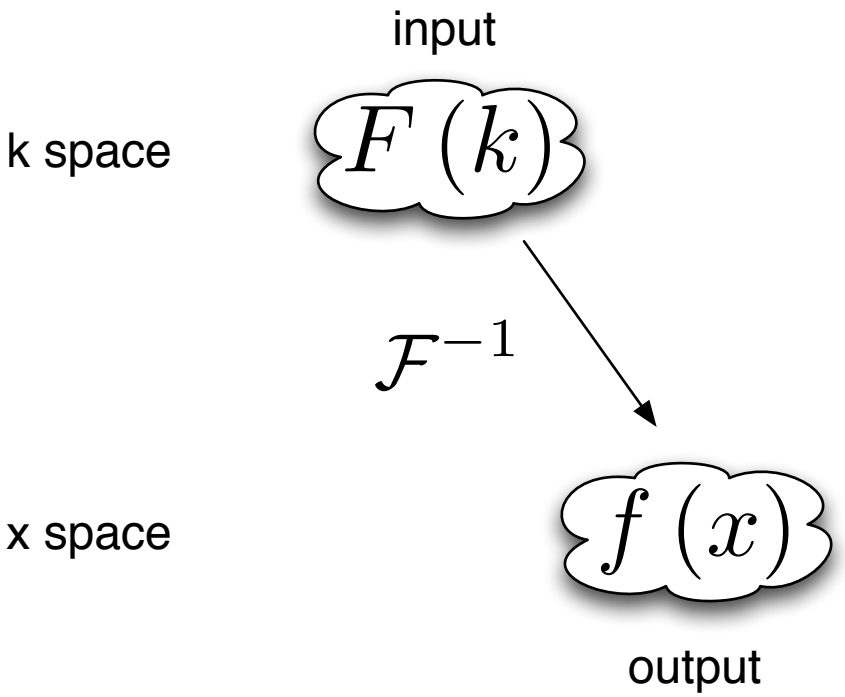
$$\mathcal{F}\left[f(x)\right](k) = \frac{1}{2\pi}\int_{x_{\min}}^{x_{\max}} f(x)e^{-ixk}dx$$

The XMDS2 Fourier transform is

$$\begin{aligned}\tilde{\mathcal{F}}\left[f(x)\right](k) &= \frac{1}{2\pi}\int_{x_{\min}}^{x_{\max}} f(x)e^{-i(x+x_{\min})k}dx \\ &= e^{-ix_{\min}k}\mathcal{F}\left[f(x)\right](k)\end{aligned}$$

When the number of forward Fourier transforms and backwards Fourier transforms are unequal a phase factor is required. Some examples of using Fourier transforms in XMDS2 are shown below.

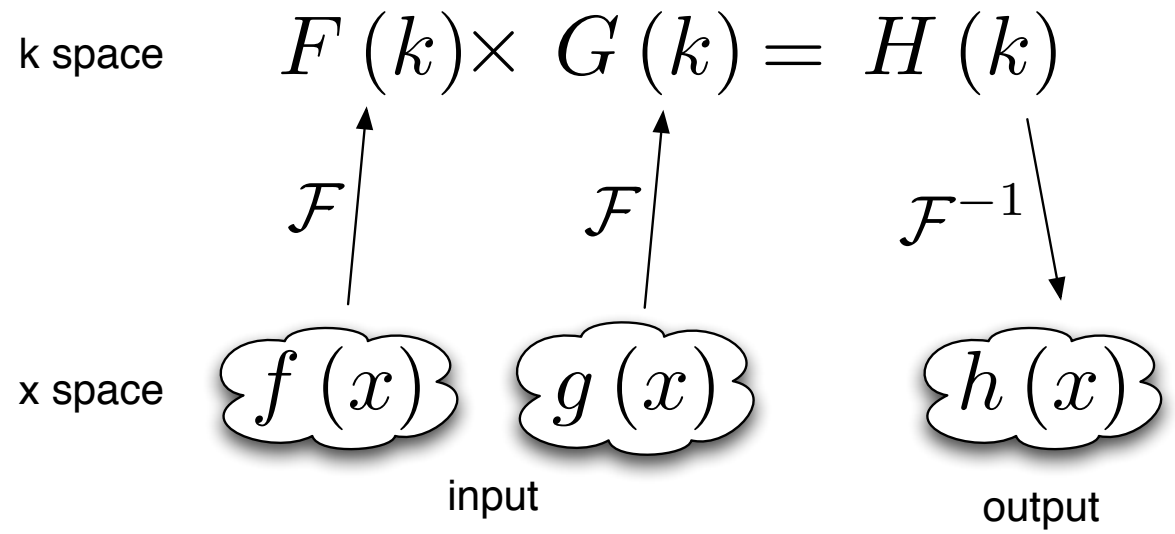
6.2.1 Example 1



When data is input in Fourier space and output in real space there is one backwards Fourier transform is required. Therefore the Fourier space data must be multiplied by a phase factor before the backwards Fourier transform is applied.

$$\mathcal{F}^{-1}[F(k)](x) = \tilde{\mathcal{F}}[e^{ix_{\min}k}F(k)](x)$$

6.2.2 Example 2

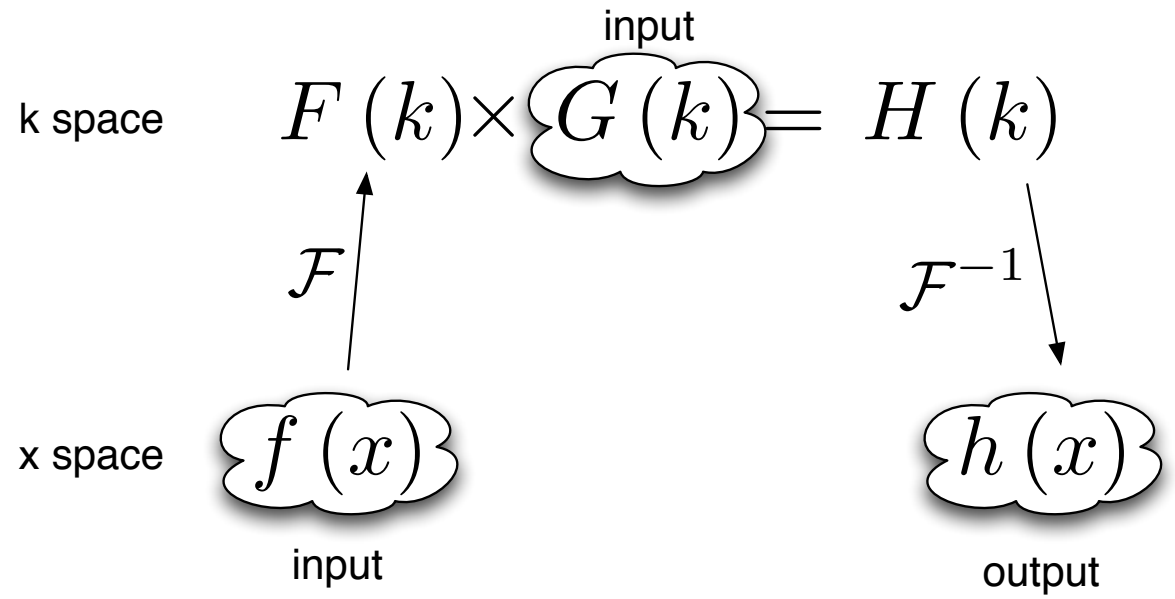


Functions of the form $h(x) = \int f(x')g(x - x')dx'$ can be evaluated using the convolution theorem:

$$\mathcal{F}[h(x)](k) = \mathcal{F}[f(x)](k) \times \mathcal{F}[g(x)](k)$$

This requires two forward Fourier transforms to get the two functions f and g into Fourier space, and one backwards Fourier transform to get the resulting product back into real space. Thus in Fourier space the product needs to be multiplied by a phase factor $e^{-ix_{\min}k}$

6.2.3 Example 3



Sometimes when the convolution theorem is used one of the forward Fourier transforms is calculated analytically and input in Fourier space. In this case only one forward numerical Fourier transform and one backward numerical Fourier transform is used. The number of forward and backward transforms are equal, so no phase factor is required.

6.3 'Loose' `geometry_matching_mode`

6.4 Dimension aliases

Dimension aliases specify that two or more dimensions have exactly the same `lattice`, `domain` and `transform`. This can be useful in situations where the problem enforces this, for example when computing correlation functions or representing square matrices.

Dimension aliases are not just a short-hand for defining an additional dimension, they also permit dimensions to be accessed *non-locally*, which is essential when computing spatial correlation functions.

Here is how to compute a spatial correlation function $g^{(1)}(x, x') = \psi^*(x)\psi(x')$ of the quantity `psi`:

```
<simulation xmds-version="2">

  <!-- name, features block -->

  <geometry>
    <propagation_dimension> t </propagation_dimension>
    <transverse_dimensions>
      <dimension name="x" lattice="1024" domain="(-1.0, 1.0)" aliases="xp" />
    </transverse_dimensions>
  </geometry>

  <vector name="wavefunction" type="complex" >
    <components> psi </components>
    <initialisation>
      <!-- initialisation code -->
    </initialisation>
  </vector>

  <computed_vector name="correlation" dimensions="x xp" type="complex" >
    <components> g1 </components>
    <evaluation>
      <dependencies> wavefunction </dependencies>
      <![CDATA[
        g1 = conj(psi(x => x)) * psi(x => xp);
      ]]>
    </evaluation>
  </computed_vector>

  <!-- integration and sampling code -->

</simulation>
```

In this simulation note that the vector `wavefunction` defaults to only having the dimension "x" even though "xp" is also a dimension (implicitly declared through the `aliases` attribute). `vector`'s without an explicit `dimensions` attribute will only have the dimensions that are explicitly listed in the `transverse_dimensions` block, i.e. this will not include aliases.

See the example `groundstate_gaussian.xmds` for a complete example.

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS

7.1 XMDS scripts look complicated! How do I start?

If you're unfamiliar with XMDS2, writing a script from scratch might seem difficult. In most cases, however, the best approach is to take an existing script and modify it for your needs. At the most basic level, you can simply take a script from the /examples directory that is similar to what you want to do, change the name of the integration variable(s) and replace the line describing the differential equation to use your DE instead. That's all you need to do, and will ensure all the syntax is correct and all the required XML blocks are present.

You can then incrementally change things such as the number of output points, what quantities get output, number of grid points, and so on. Many XMDS2 users have never written a script from scratch, and just use their previous scripts and example scripts as a scaffold when they create a script for a new problem.

7.2 Where can I get help?

The documentation on this website is currently incomplete, but it still covers a fair bit and is worth reading. Similarly, the example scripts in the /examples directory cover most of the functionality of XMDS2, so it's worth looking looking through them to see if any of them do something similar to what you're trying to do.

You should also feel free to email questions to the XMDS users' mailing list at xmds-users@lists.sourceforge.net, where the developers and other users can assist you. You can join the mailing list by going to http://sourceforge.net/projects/xmds/ and clicking on "mailing lists." Also, if you look through the mailing list archives, your particular problem may already have been discussed.

7.3 How should I cite XMDS2?

If you publish work that has involved XMDS2, please cite it as: Comput. Phys. Commun. 184, 201-208 (2013).

7.4 I think I found a bug! Where should I report it?

Please report bugs to the developer mailing list at xmds-devel@lists.sourceforge.net. In your email, please include a description of the problem and attach the XMDS2 script that triggers the bug.

7.5 How do I put time dependence into my vectors?

Standard vectors can't have time dependence (or, more accurately, depend on the `propagation_dimension` variable), but computed vectors can. So, for example, if you have set your `propagation_dimension` as "t", you can simply use the variable "t" in your computed vector and it will work.

Alternatively, you can explicitly use the `propagation_dimension` variable in your differential equation inside the `<operators>` block.

7.6 Can I specify the range of my domain and number of grid points at run-time?

Yes, you can. In your script, specify the domain and number of grid points as arguments to be passed in at run-time, use those variables in your `<geometry>` block rather than explicitly specifying them, and use the `<validation kind="run-time" />` feature. See the *Validation* entry in the Reference section for an example.

While the domain can always be specified in this way, specifying the lattice size at run-time is currently only allowed with the following transforms: 'dct', 'dst', 'dft' and 'none' (see *Transforms* in the Reference section).

Also note that for some multi-dimensional spaces using different transforms, XMDS2 will sometimes optimise the code it generates based on the relative sizes of the dimensions. If one or more of the lattices are specified at run-time it is unable to do this and will have to make guesses. In some situations this may result in slightly slower code.

7.7 When can I use IP operators (and why should I) and when must I use EX operators?

An *<operator>* that specifies named operators to be used in integration equations can have the `kind="IP"` or `kind="EX"` attribute, standing for 'interaction picture' and 'explicit' operators respectively. Explicit operators can be used in all situations, and simply construct and calculate a new vector of the form in the square brackets. IP operators use less memory and can improve speed by allowing larger timesteps, but have two important restrictions. **Use of IP operators without understanding these restrictions can lead to incorrect code**.

Some explanation is in order. The IP algorithm applies the operator separately to the rest of the evolution. The reason this can be so effective is that the separate evolution can be performed exactly. The solution of the equation $\frac{d\psi}{dt} = L\psi$ is $\psi(t + \Delta t) = exp(L\Delta t)\psi(t)$ for arbitrarily large timestep Δt. For a diagonal linear `L`, the matrix exponential is straightforward. Also, when it is constant, then the exponential can be computed and stored prior to the integration, which makes the implementation of this operator very cheap. Thus, when IP operators are defined, XMDS2 reads the equations as written by the user, and determines which operators to apply to which fields. It then implements these operators separately, and the text describing the operator inside the equations (in this example, the `L[psi]` term) is replaced by the numeral zero.

Therefore, the limitations of IP operators themselves means that they can only be applied to to named components of one of the integration vectors, and not functions of those components. Furthermore, an IP operator acting on a component must only be used in the derivative for that particular component. Secondly, due to the implementation of IP operators in XMDS2, it is not safe to use them in comments, or in conjunction with declared variables. It is also not safe to multiply or divide them by any factors, functions or vectors. They must turn up in a purely additive way when defining the derivative of a component of an integration vector. The XMDS2 parser attempts to catch possible violations of these rules, and will produce warnings in some cases.

7.8 Visual Editors

In this section goes stuff about how to set up TextMate (or other editors to highlight xpdeint scripts).

CHAPTER

EIGHT

UPGRADING FROM XMDS 1.X

While **XMDS2** is a complete rewrite of the **XMDS** project, much of the syntax has remained very similar. That said, your code will have to be rewritten as an XMDS2 program. We recommend that you work through the *Quickstart Tutorial* and perhaps the *Worked Examples* sections, and then you should be good to go.

The main news when switching to XMDS2 is the long list of new things you can do. If it's an initial value problem, XMDS2 has a good chance of being able to solve it.

We have made the decision to call the executables "xmds2" and "xsil2graphics2" so that you can keep using your old installation in parallel with the new version.

CHAPTER

NINE

OPTIMISATION HINTS

There are a variety of things you can do to make your simulations run faster.

9.1 Geometry and transform-based tricks

9.1.1 Simpler simulation geometries

Consider symmetry, can you use `dct` transforms or `bessel` transforms? Do you really need that many points? How big does your grid need to be? Could absorbing boundary conditions help?

9.1.2 Tricks for Bessel and Hermite-Gauss transforms

Dimensions using matrix transforms should be first for performance reasons. Unless you're using MPI, in which case XMDS can work it out for the first two dimensions. Ideally, XMDS would sort it out in all cases, but it's not that smart yet.

9.2 Reduce code complexity

Avoid transcendental functions like $\sin(x)$ or $\exp(x)$ in inner loops. Not all operations are made equal, use multiplication over division.

9.2.1 Use the Interaction Picture (IP) operator

Just do it. Only use the EX operator when you have to. If you must use the EX operator, consider making it `constant="no"`. It uses less memory. When you use the IP operator, make sure you know what it's doing. Do not pre- or post-multiply that term in your equations.

When using the IP operator, check if your operator is purely real or purely imaginary. If real, (e.g. `L = -0.5*kx * kx;`), then add the attribute `type="real"` to the `<operator kind="ip">` tag. If purely imaginary, use `type="imaginary"`. This optimisation saves performing the part of the complex exponential that is unnecessary.

9.2.2 Consider writing the evolution in spectral basis

The basis that makes most sense will be the one which is the 'hardest' to integrate.

9.2.3 Don't recalculate things you don't have to

Use `computed_vectors` appropriately.

9.3 Compiler and library tricks

9.3.1 Faster compiler

If you're using an Intel CPU, then you should consider using their compiler, icc. They made the silicon, and they also made a compiler that understands how their chips work significantly better than the more-portable GCC.

9.3.2 Faster libraries

Intel MKL is faster than ATLAS, which is faster than GSL CBLAS. If you have a Mac, then Apple's vecLib is plenty fast.

9.3.3 Auto-vectorisation

Auto-vectorisation is a compiler feature that makes compilers generate more efficient code that can execute the same operation on multiple pieces of data simultaneously. To use this feature, you need to add the following to the `<features>` block at the start of your simulation:

```
<auto_vectorise />
```

This will make xpdeint generate code that is more friendly to compiler's auto-vectorisation features so that more code can be vectorised. It will also add the appropriate compiler options to turn on your compiler's auto-vectorisation features. For auto-vectorisation to increase the speed of your simulations, you will need a compiler that supports it such as gcc 4.2 or later, or Intel's C compiler, `icc`.

9.3.4 OpenMP

OpenMP is a set of compiler directives to make it easier to use threads (different execution contexts) in programs. Using threads in your simulation does occur some overhead, so for the speedup to outweigh the overhead, you must have a reasonably large simulation grid. To add these compiler directives to the generated simulations, add the tag `<openmp />` in the `<features>` block. This can be used in combination with the auto-vectorisation feature above. Note that if you are using gcc, make sure you check that your simulations are faster by using this as gcc's OpenMP implementation isn't as good as icc's.

If you are using the OpenMP feature and are using FFTW-based transforms (Discrete Fourier/Cosine/Sine Transforms), you should consider using threads with your FFT's by adding the following to the `<features>` block at the start of your simulation:

```
<fftw threads="2" />
```

Replace the number of threads in the above code by the number of threads that you want to use.

9.3.5 Parallelisation with MPI

Some simulations are so large or take so much time that it is not reasonable to run them on a single CPU on a single machine. Fortunately, the Message Passing Interface was developed to enable different computers working on the same program to exchange data. You will need a MPI package installed to be abel to use this feature with your simulations. One popular implementation of MPI is OpenMPI.

9.4 Atom-optics-specific hints

9.4.1 Separate out imaginary-time calculation code

When doing simulations that require the calculation of the groundstate (typically via the imaginary time algorithm), typically the groundstate itself does not need to be changed frequently as it is usually the dynamics of the simulation that have the interesting physics. In this case, you can save having to re-calculate groundstate every time by having one script (call it `groundstate.xmds`) that saves the calculated groundstate to a file using a breakpoint, and a second simulation that loads this calculated groundstate and then performs the evolution. More often than not, you won't need to re-run the groundstate finder.

The file format used in this example is HDF5, and you will need the HDF5 libraries installed to use this example. The alternative is to use the deprecated `binary` format, however to load `binary` format data `xmds`, the predecessor to `xpdeint` must be installed. Anyone who has done this before will tell you that installing it isn't a pleasant experience, and so HDF5 is the recommended file format.

If your wavefunction vector is called `'wavefunction'`, then to save the groundstate to the file `groundstate_break.h5` in the HDF5 format, put the following code immediately after the integrate block that calculates your groundstate:

```
<breakpoint filename="groundstate_break" format="hdf5">
  <dependencies>wavefunction</dependencies>
</breakpoint>
```

In addition to the `groundstate_break.h5` file, an XSIL wrapper `groundstate_break.xsil` will also be created for use with *xsil2graphics2*.

To load this groundstate into your evolution script, the declaration of your `'wavefunction'` vector in your evolution script should look something like

```
<vector name="wavefunction">
  <components>phi1 phi2</components>
  <initialisation kind="hdf5">
    <filename>groundstate_break.h5</filename>
  </initialisation>
</vector>
```

Note that the groundstate-finder doesn't need to have all of the components that the evolution script needs. For example, if you are considering the evolution of a two-component BEC where only one component has a population in the groundstate, then your groundstate script can contain only the `phi1` component, while your evolution script can contain both the `phi1` component and the `phi2` component. Note that the geometry of the script generating the groundstate and the evolution script must be the same.

9.4.2 Use an energy or momentum offset

This is just the interaction picture with a constant term in the Hamiltonian. If your state is going to rotate like $e^{i(\omega+\delta\omega)t}$, then transform your equations to remove the $e^{i\omega t}$ term. Likewise for spatial rotations, if one mode will be moving on average with momentum $\hbar k$, then transform your equations to remove that term. This way, you may be able to reduce the density of points you need in that dimension. Warning: don't forget to consider this when looking at your results. I (Graham Dennis) have been tripped up on multiple occasions when making this optimisation.

CHAPTER

TEN

XSIL2GRAPHICS2

xsil2graphics2 is a way of converting ".xsil" files to formats that other programs can read. The syntax is described in the *Quickstart Tutorial*, and by using the `xsil2graphics2 --help` option. It currently can covert any output format for use by Mathematica.

We recommend HDF5 format instead of the binary format for output and input, as many visualisation tools can already read/write to this format directly.

CHAPTER

ELEVEN

DEVELOPER DOCUMENTATION

Developers need to know more than users. For example, they need to know about the test suite, and writing test cases. They need to know how to perform a developer installation. They need to know how to edit and compile this documentation. They need a step-by-step release process.

11.1 Test scripts

Every time you add a new feature and/or fix a new and exciting bug, it is a great idea to make sure that the new feature works and/or the bug stays fixed. Fortunately, it is pleasantly easy to add a test case to the testing suite.

1. Write normal XMDS script that behaves as you expect.
2. Add a `<testing>` element to your script. You can read the description of this element and its contents below, and have a look at other testcases for examples, but the basic structure is simple:.

```
<testing>
  <command_line> </command_line>
  <arguments>
    <argument />
    <argument />
    ...
  </arguments>
  <input_xsil_file />
  <xsil_file>
    <moment_group />
    <moment_group />
    ...
  </xsil_file>
</testing>
```

3. Put into the appropriate `testsuite/` directory.
4. run `./run_tests.py` This will automatically generate your `_expected` files.
5. Commit the `.xmds`, `*_expected.xsil` file and any `*_expected*` data files.

11.1.1 Testing element

11.1.2 command_line element

11.1.3 input_xsil_file element

11.1.4 xsil_file element

11.1.5 moment_group element

11.2 Steps to update XMDS script validator (XML schema)

1. Modify `xpdeint/support/xpdeint.rnc`. This is a RelaxNG compact file, which specifies the XML schema which is only used for issuing warnings to users about missing or extraneous XML tags / attributes.
2. Run `make` in `xpdeint/support/` to update `xpdeint/support/xpdeint.rng`. This is the file which is actually used, which is in RelaxNG format, but RelaxNG compact is easier to read and edit.
3. Commit both `xpdeint/support/xpdeint.rnc` and `xpdeint/support/xpdeint.rng`.

11.3 Directory layout

11.3.1 XMDS2's code and templates

All `.tmpl` files are Cheetah template files. These are used to generate C++ code. These templates are compiled as part of the XMDS2 build process to `.py` files of the same name. Do not edit the generated `.py` files, always edit the `.tmpl` files and regenerate the corresponding `.py` files with `make`.

- **`xpdeint/`:**
  - **`Features/`: Code for all `<feature>` elements, such as `<globals>` and `<auto_vectorise>`**
    - `Transforms/`: Code for the Fourier and matrix-based transforms (including MPI variants).
  - `Geometry/`: Code for describing the geometry of simulation dimensions and domains. Includes code for `Geometry`, `Field` and all `DimensionRepresentations`.
  - `Operators/`: Code for all `<operator>` elements, including `IP`, `EX` and the temporal derivative operator `DeltaA`.
  - **`Segments/`: Code for all elements that can appear in a `<segments>` tag. This includes `<integrate>`, `<`**
    - `Integrators`: Code for fixed and adaptive integration schemes, and all steppers (e.g. `RK4`, `RK45`, `RK9`, etc.)
  - **`Stochastic/`: Code for all random number generators and the random variables derived from them.**
    - `Generators/`: Code for random number generators, includes `dSFMT`, `POSIX`, `Solirte`.
    - `RandomVariables/`: Code for the random variables derived from the random number generators. These are the gaussian, poissonian and uniform random variables.
  - `SimulationDrivers/`: Code for all `<driver>` elements. In particular, this is where the location of MPI and multi-path code.

  - `Vectors/`: Code for all `<vector>` elements, and their initialisation. This includes normal `<vector>` elements as well as `<computed_vector>` and `<noise_vector>` elements.
  - `includes/`: C++ header and sources files used by the generated simulations.
  - **`support/`: Support files**
    * `wscript`: `waf` build script for configuring and compiling generated simulations
    * `xpdeint.rnc`: Compact RelaxNG XML validation for XMDS scripts. This is the source file for the XML RelaxNG file `xpdeint.rng`
    * `xpdeint.rng`: RelaxNG XML validation for XMDS scripts. To regenerate this file from `xpdeint.rnc`, just run `make` in this directory.
  - `waf/`: Our included version of the Python configuration and build tool `waf`.
  - `waf_extensions/`: `waf` tool for compiling Cheetah templates.
  - `xsil2graphics2/`: Templates for the output formats supported by `xsil2graphics2`.
  - `wscript`: `waf` build script for XMDS2 itself.
  - `CodeParser.py`: Minimally parses included C++ code for handling nonlocal dimension access, IP/EX operators and IP operator validation.
  - `Configuration.py`: Manages configuration and building of generated simulations.
  - `FriendlyPlusStyle.py`: Sphinx plug-in to improve formatting of XMDS scripts in user documentation.
  - This directory also contains code for the input script parser, code blocks, code indentation, and the root `_ScriptElement` class.

11.3.2 Support files

- **`admin/`: Documentation source, Linux installer and release scripts.**
  - `developer-doc-source/`: source for epydoc python class documentation (generated from python code).
  - `userdoc-source/`: source for the user documentation (results visible at www.xmds.org and xmds2.readthedocs.org).
  - `xpdeint.tmbundle/`: TextMate support bundle for Cheetah templates and XMDS scripts
- `bin/`: Executable scripts to be installed as part of XMDS2 (includes `xmds2` and `xsil2graphics2`).
- `examples/`: Example XMDS2 input scripts demonstrating most of XMDS2's features.
- `testsuite/`: Testsuite of XMDS2 scripts. Run the testsuite by executing `./run_tests.py`

CHAPTER

TWELVE

LICENSING

XMDS2 is licensed under the GPL version 2 license, which can be found in the COPYING file in the root directory of your XMDS install. You can also find it here: http://www.gnu.org/licenses/

We encourage people to submit patches to us that extend XMDS2's functionality and fix bugs. If you do send us a patch, we do not require copyright assignment, but please include a statement saying you agree to license your code under the GPL v2 license.