# xmds:

e X tensible

M ulti

D imensional

S imulator

An open source numerical simulation package.

Version 1.6.5

P. T. Cochrane, G. Collecutt, P. D. Drummond, and J. J. Hope

May 27, 2012

# Abstract

Writing codes for the simulation of complex phenomena is an art and science unto itself. What with finding and using good algorithms, actually writing the code, debugging the code and testing the code, not much time is left to actually investigate what it was you were initially out to look at. This is where **xmds** comes in. **xmds** allows you to write a high-level description of the problem you are trying to solve (usually a differential equation of some form) it goes away and writes low-level simulation code for you (trying very hard to keep the code as efficient as possible), compiles it and presents it, ready to be run.

**xmds** is an acronym for eXtensible Multi-Dimensional Simulator. It is open source software, released under the GNU General Public License, and is written to assist in the calculation of a wide range of problems, with application in physics, mathematics, and even finance and economics. A high-level description of the problem at hand is written in XML (the extensible markup language) and **xmds** transforms this into C language code. This code can then be compiled by a C/C++ compiler to produce a binary executable which solves the problem about as quickly and efficiently as might be achieved with code written by an expert.

**xmds** gives modellers—people doing computer modelling—structure, organisation and standardisation. It provides a framework for describing the system they are trying to simulate, be it in a physical, mathematical, scientific or even financial or economic setting. It gives a way to keep the ideas behind a simulation well laid out and, importantly, documented for others to see and use. And it gives people a common ground from which they can compare their numerical work; something desperately lacking in an area at the interface between theory and experiment, which already have a well-ingrained culture of comparison and verification.

## About this manual

This manual has been split into five parts in an attempt to cover all of the material necessary to be able to use and master **xmds**, but also to provide an entry point for novices and experts alike. For the impatient, it may be possible to learn jump right past the tutorial, and learn what you need to get started from the worked examples in section 9.

Part I is a very simple introduction to **xmds**, and discusses how to build a simulation script from scratch, how to modify existing code or a template to perform the simulation you want, and how to perform stochastic simulations using the Message Passing Interface (MPI). Novice users may wish to start with Chapter 1 (Starting from scratch) or Chapter 3 (Using a template) to help themselves get going with **xmds**. Advanced users of **xmds** may

skip these chapters, and may be more interested in how to use **xmds** with MPI for running parallel simulations, which is discussed near the end of Part I.

Part II is a reasonably general discussion of numerical techniques for solution of differential equations. Some of the techniques discussed are used within **xmds**, and the procedures used internally in **xmds** are outlined.

Part III is an overview of the language and contains some of the details of working with and the workings of **xmds**. This along with Part II is the essence of the `xmds-1.0` documentation. In time this Part will be updated, but at present only superficial changes have been made in its layout.

Part IV gives specific information about the keywords used in the **xmds** language, their place within a script's structure, examples of usage, and what **xmds** expects as arguments within the respective tags. This Part is particularly aimed at users familiar with **xmds** who are likely to want to know the syntax of a particular keyword or other details of a particular keyword.

Part V is an appendix and covers the XSIL output format, the `xsil2graphics` utility program, the `loadxsil.m` script, the GNU General Public License, and a bibliography.

# Tools used to build xmds

These are the multifarious tools with which **xmds**, its documentation (both handmade and automatically generated) and its web pages, has been made.

- GNU development suite: autoconf, automake, etc.
- General development tools: make, gcc, g++, cvs
- Editors: emacs, vim
- Linux Distributions: Gentoo Linux, Redhat Linux
- Other Unix and Unix-like environments: True64, CygWin
- Scripting tools and languages: aap, perl, python
- Documentation tools: doxygen, LaTeX, latex2html
- Libraries: fftw
- Organisations: Sourceforge.net, APAC

# Feedback

Yes, we want feedback! If you have any comments about **xmds** and/or this manual (such as, inaccuracies, possible improvements, new features, what it does well, etc.) then please email one of the current developer or the **xmds** web page webmaster. You can find the addresses of both of these people on the **xmds** web page: http://www.xmds.org. And please, feel free

to mention anything, no matter how small. It would be great to see **xmds** improve the way people want, and for it to be documented the way the **xmds** user community wants.

# Contents

# List of Figures

# List of Tables

# Part I

# Tutorial

# 1 Starting from scratch

It turns out that one of the most difficult things to do in **xmds** is to write a script from scratch, which is why most users take either one of their own old scripts, or borrow someone else's (or one of the examples) as a template. However, this doesn't mean to say that one is never going to have to write a script from scratch (just that it's usually quite unlikely), therefore we explain how to do this here. If you're not interested, and want to get coding more quickly, then it's alright to skip to Chapter 3 and learn how to modify an existing script to your needs.

In this chapter we will outline the tags necessary for **xmds** to actually parse a document, and will implement a simple simulation involving these tags. More complete documentation of the tags and what they do can be found in Chapter 11. The tags necessary for performing more advanced operations and for solving more complex problems will be introduced in later tutorial chapters (e.g. see Chapter 4).

## 1.1 A basic simulation

### 1.1.1 The simple beginnings of a simulation

**xmds** is coded using XML (the extensible markup language), and as such each **xmds** script must be a "properly formed" XML document. One of the main stipulations for an XML document to be properly formed is that it have the following line at the top of each document, and so, each **xmds** script **must** have this as its first line:

```
<?xml version="1.0"?>
```

There are other stipulations, but they don't really concern us too much at the moment. If you're interested, you can check out the World Wide Web Consortium (W3C) web site (http://www.w3c.org) for more information.

Each **xmds** simulation is enclosed within a set of `<simulation>` tags. Therefore, for each simulation that you write from scratch, the first few lines of code are going to be:

```
<?xml version="1.0"?>
<simulation>

  <!-- yet more xmds code to come -->

</simulation>
```

It might be a good idea at this stage to save this code to file. So, for the purposes of the tutorial, we'll refer to this script as `lorenz.xmds`.

Next, the **xmds** script needs a name. Well, to be honest, it doesn't really need a name, but it's a good idea to add the `<name>` tag if only for completeness. If the `<name>` tag is not included, **xmds** uses the name of the **xmds** script file (but with the `.xmds` extension removed) as the name of the simulation, so in this simulation if we were to not specify the `<name>` then the simulation name used by **xmds** would be `lorenz`. However, it is still a good idea to specify the name inside the simulation text and we do this here, setting `<name>` to `lorenz`. The script now becomes:

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>

  <!-- yet more xmds code to come -->

</simulation>
```

Notice that we have indented the `<name>` tag from the rest of the tags. It is a good idea to indent tags that are nested within other tags so that one gets an idea of the document structure just from looking at the source code.

The next two tags that I recommend you add to your scripts are the `<author>` and `<description>` tags. These tags aren't necessary for running a simulation or for actually calculating anything, however, they are good for documenting the simulation, and providing extra information that may be helpful to others if you show your scripts to other people. Also, this information may actually be used in future versions of **xmds** to provide extra functionality in the output from **xmds** simulations. The code now is:

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>
```

```
  <!-- yet more xmds code to come -->

</simulation>
```

Alternatively, you can add such information to the script by putting comments in the source code. The above code may then look something like this:

```
<?xml version="1.0"?>
<!-- Example simulation: lorenz -->

<!-- Adapted from the example in "Numerical methods for physics"-->
<!-- by Alejandro L. Garcia, pg 78-->

<!-- Xmds script by: Paul Cochrane -->

<simulation>

  <name>lorenz</name>

  <!-- yet more xmds code to come -->

</simulation>
```

### 1.1.2 General simulation options

Now that the very basic preliminaries are out of the way, we need to focus on the problem we are trying to solve. So, for the sake of argument, let's try to solve the Lorenz equations,

$$\frac{dx}{dt} = \sigma(y - x) \tag{1.1}$$

$$\frac{dy}{dt} = rx - y - xz \tag{1.2}$$

$$\frac{dz}{dt} = xy - bz, \tag{1.3}$$

where $\sigma$, $r$ and $b$ are positive constants, and the variables have the initial conditions: $x(t = 0) = x_0$, $y(t = 0) = y_0$, $z(t = 0) = z_0$.

Even though we are trying to solve something as complex as a chaotic system, this is very easy to write down in **xmds**. This is especially true since the derivatives are with respect to time, for if we had spatial derivatives we would have to use Fourier transform techniques and mappings to simplify the calculation (we'll cover this stuff in Section 1.2, so don't worry that we're not discussing it here) which would complicate our script a bit more, and we're trying to keep things simple here.

The Lorenz model can be used in the study of many interesting phenomena, however it is possibly best known as a model of global weather [1]. For a system to be chaotic it must be extremely sensitive to initial conditions such that any small perterbation of the initial

conditions will cause wildly divergent evolution; and this is something we will hopefully see here, so let's continue.

There can be many global parameters in an **xmds** simulation, although, one in particular is special. This is the propagation direction, specified by the `<prop_dim>` tag, which is specified as part of the global functionality of the **xmds** simulation and appears within the `<simulation>` tags, normally after the `<description>`. In the problem we are trying to solve, our field is evolving in *time*, given by the variable $t$ in the above equations, and so the field is said to be propagating in $t$ so we use time as the propagation dimension. Therefore, we add the line

```
<prop_dim>t</prop_dim>
```

to our **xmds** script, just after the `<description>` or conversely just before the `<globals>` section (in the situation considered here, these locations are one in the same, however, this is not the case in general). The script is now

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>

  <prop_dim>t</prop_dim>

  <!-- yet more xmds code to come -->

</simulation>
```

This problem we are trying to solve has several constants, namely $\sigma$, $r$, $b$, $x_0$, $y_0$, and $z_0$. These variables are going to be used again and again in the simulation therefore it makes sense to put them into the next element necessary to describe a simulation in **xmds**, the `<globals>` tag. We specify these constants using C/C++ syntax in an XML `CDATA` block, which is enclosed within the `<globals>` element. The **xmds** code for this is

```
<globals>
<![CDATA[
    const double sigma = 10.0;
    const double b = 8.0/3.0;
    const double r = 28.0;
    const double xo = 1.0;      // initial conditions
    const double yo = 1.0;      // initial conditions
    const double zo = 20.0;     // initial conditions
]]>
</globals>
```

There are a couple of points to note here. Firstly, and most importantly, the syntax of the code within the `CDATA` block **must** conform to C/C++ syntax rules, otherwise the simulation won't be able to be compiled. This is because the code within the `CDATA` block is inserted directly into the code for the output simulation. Secondly, the constants that we are specifying are declared to be double precision to **xmds** (this is via the `double` keyword in the code above) since they are continuous variables in the problem being solved. If for instance, we had some discrete quantity such as the number of particles in a system, then we would specify the variable as being integer and would therefore use the `int` keyword to declare the variable as such. Lastly, the odd-looking tags for the `CDATA` block must be written correctly (i.e. opened with `<![CDATA[` and closed with `]]>`) otherwise the file won't parse, and **xmds** will give an error. Our script is now,

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>

  <prop_dim>t</prop_dim>

  <globals>
  <![CDATA[
      const double sigma = 10.0;
      const double b = 8.0/3.0;
      const double r = 28.0;
      const double xo = 1.0;      // initial conditions
      const double yo = 1.0;      // initial conditions
      const double zo = 20.0;     // initial conditions
  ]]>
  </globals>

  <!-- yet more xmds code to come -->

</simulation>
```

### 1.1.3 The field element

Now that we have set up the physical constants of our problem, we need to describe the field that we are going to be integrating over, in other words we need to specify a discretised version of $x(t)$, $y(t)$ and $z(t)$ at the start of the problem. To specify the field for the simple example we are studying here, we only need to tell **xmds** the initial value of the field (i.e. that

when $t = 0$). Note that in more complex situations (to be studied later) there are more tags to worry about.

The first tag is the `<field>` element. This is a container for the other information that we are using to describe the field, and is used very simply as follows

```
<field>

  <!-- More xmds tags in here -->

</field>
```

We next give the name of the field, this is supplied with the `<name>` tag (note that this is a sub-element of the `<field>` tag, and so is different from the `<name>` tag of the `<simulation>` element). If no name is given for the field, it defaults to "`main`", however, as mentioned before, it is a good idea to specify a name for the field anyway. We'll call it `main` to be consistent with other scripts you are likely to see, and because this is the main field.

We next must tell **xmds** which of the field moments to sample *directly after* the field is initialised. This is not obvious to those new to **xmds**, however this gives one the opportunity to choose whether or not to sample the initial point of the field, and generate output moments such as mean and standard error, before it is propagated. To do this we put a sequence of `1`'s or `0`'s within the `<samples>` element. We must put as many `1`'s and `0`'s as there are moment groups defined in the `<output>` tag (to be discussed later). In our case, we will only be using one output moment group here, and we do want to sample the initial field for the moments, hence we add the code

```
<samples>1</samples>
```

to our script. The simulation at this stage looks like

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>

  <prop_dim>t</prop_dim>

  <globals>
  <![CDATA[
      const double sigma = 10.0;
      const double b = 8.0/3.0;
      const double r = 28.0;
      const double xo = 1.0;      // initial conditions
      const double yo = 1.0;      // initial conditions
```

```
      const double zo = 20.0;     // initial conditions
  ]]>
  </globals>

  <field>
    <name>main</name>
    <samples>1</samples>

    <!-- yet more xmds code to come -->

  </field>

</simulation>
```

Now we have to initialise the field. In other words, we have to define $x(t = 0)$, $y(t = 0)$, and $z(t = 0)$. **xmds** makes this easy by allowing you to define the field as a vector written in terms of the dimensions of the field. It is possible to define other vectors that are part of the field, but the vector of the field that we are integrating is the *main* vector, and this we name (funnily enough) `main`. This naming is compulsory since it is possible to have more than one vector named within a field; however, there must be exactly one vector named `main`. We also need to specify the data type of our vector, the names of the components of the vector and we need to define how the vector should be calculated using C code (in a `CDATA` section). For the case we are considering here, the **xmds** code would be:

```
<vector>
  <name> main </name>
  <type> double </type>
  <components> x y z </components>
  <![CDATA[
      x = xo;
      y = yo;
      z = zo;
  ]]>
</vector>
```

So, as we can see, our vector is called `main`, it is of type `double` and the names of its components are `x`, `y`, and `z`. The `CDATA` section gives the C code version of what Equation (1.3) describes.

This code completes the `<field>` element and we are left with the following code listing:

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
```

```
  </description>

  <prop_dim>t</prop_dim>

  <globals>
  <![CDATA[
      const double sigma = 10.0;
      const double b = 8.0/3.0;
      const double r = 28.0;
      const double xo = 1.0;      // initial conditions
      const double yo = 1.0;      // initial conditions
      const double zo = 20.0;     // initial conditions
  ]]>
  </globals>

  <field>
    <name>main</name>
    <samples>1</samples>

    <vector>
      <name> main </name>
      <type> double </type>
      <components> x y z </components>
      <![CDATA[
          x = xo;
          y = yo;
          z = zo;
      ]]>
    </vector>

  </field>

  <!-- yet more xmds code to come -->

</simulation>
```

### 1.1.4 The sequence element

We have arrived at the stage where we can tell **xmds** how to actually perform the integration of the field. To do this we use the `<sequence>` element. The `<sequence>` element is usually used as a container for other elements, specifically the `<integrate>`, `<filter>` and other `<sequence>` elements. The outermost `<sequence>` element is referred to as the "parent" `<sequence>` and the `<sequence>`s nested within that as the "child" `<sequence>`s. This may sound a bit confusing, but it is just a generalisation and a lot of the time you will be writing scripts with just the one `<sequence>`. The `<sequence>` element may have as many of the other sub-elements as desired to perform the calculation, and the "child" `<sequence>`s

can contain another element—the <cycles> element—which controls how many times a given <sequence> is repeated. The <cycles> element is optional and defaults to one. It is important to note that the order of segments specified within a <sequence> are significant, and operations given will be performed in that order.

So, to summarise, most of the time you will just use one <sequence> and it will usually only contain just the one <integrate> section, hence the code will look like

```
<sequence>
  <integrate>

    <!-- More xmds tags to come -->

  </integrate>
</sequence>
```

We shall try to discuss the other features and tags in more depth later on in more advanced tutorials.

### 1.1.5 The integrate element

Since the <integrate> element is quite complex, and it does all of the hard work, we'll spend some time discussing it.

We need to tell **xmds** the algorithm to use to integrate the field specified earlier. To do this we use the <algorithm> tag. This tag is optional and will default to SIEX for stocastic simulations and to RK4EX for non-stochastic simulations, however, it is a very good idea to explicitly specify what algorithm your simulation is using, if only to help yourself in six months time, or a colleague who may end up reading your code. At the moment (**xmds** version 1.5-1) there are six algorithms to choose from: RK4EX, RK4IP, ARK45EX, ARK45IP, SIEX, SIIP. RK4EX is a fourth order Runge-Kutta in the explicit picture, RK4IP is a fourth order Runge-Kutta in the interaction picture, the ARK45EX and ARK45IP are the corresponding adaptive time step Runge-Kutta Fehlberg methods, SIEX is the semi-implicit method in the explicit picture, and SIIP is the semi-implicit method in the interaction picture. For more information about the specifics of these algorithms and techniques, see Section 6.3. In solving our problem we'll use the fourth order Runge-Kutta in the explicit picture, because we aren't using any Fourier transforms, and the explicit picture is fine for our purposes here. Therefore, we specify within the <integrate> tags the line:

```
<algorithm>RK4EX</algorithm>
```

telling **xmds** what algorithm to use.

The next things **xmds** needs to know are the length of the integration interval, the total number of steps to take, and the number of samples for each output moment to take within these steps. These items are denoted by the <interval>, <lattice> and <samples> tags respectively. The integration interval combined with the number of steps gives the step size internally used by **xmds**. We'll choose some fairly arbitrary numbers here: an interval length of 10, a large number of lattice points, namely 10000, (which should give us a nice small step size), and we'll sample 200 points, and hence set <samples> to 200. The code for this looks like:

```
<interval> 10 </interval>
<lattice> 10000 </lattice>
<samples> 200 </samples>
```

Having told **xmds** some of the parameters it must use to perform the integration, we haven't yet told it *how* to actually carry out the integration. By this I mean that we have to describe in terms of C language code the differential equation that **xmds** is to use to evolve the solution forward (see Equation (1.3)). We do this by using a `CDATA` block, and writing the equation in a form understandable to both us and **xmds** which isn't technically speaking C code.

```
<![CDATA[
    dx_dt = sigma*(y - x);
    dy_dt = r*x - y - x*z;
    dz_dt = x*y - b*z;
]]>
```

Notice that this looks similar to the analytical form of Equation (1.3) in that we have described the derivatives with respect to time, $t$ of the fields $x(t)$, $y(t)$ and $z(t)$.

With that, we have completed the `<integrate>` element, and the `<sequence>` section. The simulation script is now:

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>

  <prop_dim>t</prop_dim>

  <globals>
  <![CDATA[
      const double sigma = 10.0;
      const double b = 8.0/3.0;
      const double r = 28.0;
      const double xo = 1.0;      // initial conditions
      const double yo = 1.0;      // initial conditions
      const double zo = 20.0;     // initial conditions
  ]]>
  </globals>

  <field>
    <name>main</name>
    <samples>1</samples>
```

```
    <vector>
      <name> main </name>
      <type> double </type>
      <components> x y z </components>
      <![CDATA[
          x = xo;
          y = yo;
          z = zo;
      ]]>
    </vector>
  </field>

  <sequence>
    <integrate>
      <algorithm>RK4EX</algorithm>
      <interval> 10 </interval>
      <lattice> 10000 </lattice>
      <samples> 200 </samples>
      <![CDATA[
          dx_dt = sigma*(y - x);
          dy_dt = r*x - y - x*z;
          dz_dt = x*y - b*z;
      ]]>
    </integrate>
  </sequence>

  <!-- yet more xmds code to come -->

</simulation>
```

### 1.1.6 The output element

Ok, we're almost there! This is the last section that we need to worry about. So, just to recap, we've told **xmds** the general features and variables it should use to construct the simulation, the field to integrate over, and the way in which the integration should take place and most importantly the differential equation that **xmds** should use to evolve the solution. That sounds like about it doesn't it? Well, no. We haven't told **xmds** to output anything yet, and it's a bit silly to spend several hours of computer time for the simulation to come back to you and say: "I'm done!" and you haven't got any results. Fortunately, **xmds** doesn't let you write a simulation without actually specifying any output. Therefore, we need to tell **xmds** what output we want from the simulation, and to do this we use the `<output>` element.

The `<output>` element is just a container for the other tags that specify what is to be output. The two tags that are contained within the `<output>` element are: `<filename>` and `<group>`.

The <filename> tag (fairly obviously) specifies the filename of the output data file. This tag is optional and defaults to the simulation name (i.e. the value of <name> directly within the <simulation> tag) with the string .xsil appended. For example, if we didn't specify a filename for the simulation we've created here, then the filename **xmds** would use would be basicSim.xsil. The output data file is in the XSIL format [2] which is a handy interchange format also using XML.

The <group> tag contains a description (and to a degree the definition) of the moments of the output data, which can be such things as the power density of the field(s), or just means and standard errors of the field(s). More than one output group can be specified, but at least one must be given. Post-processing may be performed before an output group is written to file, allowing one to do some complex tasks on the data sampled in the running of the simulation (this is put after the <sampling> tag which is coming up), however this is more involved than the discussion here, so we won't be using it. A good place to look if you're at all interested is the examples directory in the distribution or see the script repository on the **xmds** web page: http://www.xmds.org.

After looking for the <group> tag, **xmds** then expects to see a <sampling> tag within that, which defines how the group is to be sampled. This is also just a container for more specific tags. Just as an aside: although it may seem a pain at this stage to have so many containers for other containers and tags and so on, this gives the entire document a nice structure where one builds up a simulation from nice bite-sized (pun not intended) chunks. Also, one really doesn't go to the pain of writing a simulation from scratch very often so this shouldn't be a big issue when you finally get to writing other and more complex (and more interesting!) simulations. Within the <sampling> tag **xmds** expects to see a <fourier_space> tag for each transverse dimension in the simulation, which tells **xmds** whether or not to Fourier transform the dimension before being sampled and written out to disk. For our example here, we don't have any transverse dimensions so we won't use this tag.

Now we need to tell **xmds** the names of output moments we want to calculate, and the C code it should use to do so. We do this using the <moments> tag and a CDATA block. For this simulation things are quite simple since we just want the amplitude of the variables as they evolve. Just to make this really obvious we'll define the output moments to be new variables called xOut, yOut and zOut which are just equal to the variables x, y and z, but it makes it more obvious in the code what information we are grabbing out of **xmds**. It is possible to define as many moments as you wish, but you must define at least one. The XML code we use is

```
<sampling>
  <moments> xOut yOut zOut </moments>
  <![CDATA[
      xOut = x;
      yOut = y;
      zOut = z;
  ]]>
</sampling>
```

giving an output section which looks like

```
<output>
  <sampling>
    <moments> xOut yOut zOut </moments>
    <![CDATA[
        xOut = x;
        yOut = y;
        zOut = z;
    ]]>
  </sampling>
</output>
```

and an overall simulation which is (finally):

```
<?xml version="1.0"?>
<simulation>

  <name>lorenz</name>
  <author>Paul Cochrane</author>
  <description>
    Lorenz attractor example simulation.  Adapted from the example
    in "Numerical methods for physics" by Alejandro L. Garcia,
    page 78 (1st ed.).
  </description>

  <prop_dim>t</prop_dim>

  <globals>
  <![CDATA[
      const double sigma = 10.0;
      const double b = 8.0/3.0;
      const double r = 28.0;
      const double xo = 1.0;      // initial conditions
      const double yo = 1.0;      // initial conditions
      const double zo = 20.0;     // initial conditions
  ]]>
  </globals>

  <field>
    <name>main</name>
    <samples>1</samples>

    <vector>
      <name> main </name>
      <type> double </type>
      <components> x y z </components>
      <![CDATA[
          x = xo;
          y = yo;
```

```
            z = zo;
        ]]>
      </vector>
    </field>

    <sequence>
      <integrate>
        <algorithm>RK4EX</algorithm>
        <interval> 10 </interval>
        <lattice> 10000 </lattice>
        <samples> 200 </samples>
        <![CDATA[
            dx_dt = sigma*(y - x);
            dy_dt = r*x - y - x*z;
            dz_dt = x*y - b*z;
        ]]>
      </integrate>
    </sequence>

    <output>
      <sampling>
        <moments> xOut yOut zOut </moments>
        <![CDATA[
            xOut = x;
            yOut = y;
            zOut = z;
        ]]>
      </sampling>
    </output>
</simulation>
```

Here is a link to the finished (gzipped) script file lorenz.xmds.gz on the **xmds** web site (http://www.xmds.org).

### 1.1.7 Making the simulation and getting results

Now that the simulation script is ready, it is just a matter of getting **xmds** to generate the C++ source code and compiling that with your system's C++ compiler. This is a very simple process, and in the vast majority of cases, all one has to do is enter the following at the command prompt:

```
% xmds lorenz.xmds
```

A file called `lorenz` should now appear in the same directory as your simulation script; this file is the simulation binary executable file. To run the simulation merely execute the binary file by entering its name at the command line. You should see something like this

```
% lorenz
Beginning full step integration ...
```

```
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 5.000000e-02
<snip>
Sampled field (for moment group #1) at t        = 9.950000e+00
Sampled field (for moment group #1) at t        = 1.000000e+01
maximum step error in moment group 1 was 1.248578e-05
```

Once the program has finished running, you should find in the same directory as the binary executable a file called `lorenz.xsil`. This is the file containing your output data in a handy XML based format that can be used to interchange data between various other formats. We'll look at the output here in two programs, namely Matlab (or Octave) and Scilab. Matlab is a commercial numerical programming language and environment which has very powerful graphics capabilities and is used in the scientific community extensively. Octave is a free program which is highly compatible with Matlab. Scilab is very similar to Matlab, however it is free to download and install, but doesn't have quite the same quality as that made by Matlab. Nevertheless, Scilab is free, and is a handy alternative if your budget can't stretch to Matlab. There are subtle differences between Matlab (or Octave) and Scilab and this is why we discuss the two here. XSIL files can also easily be translated into scripts suited for input into Mathematica, gnuplot, or R using the bundled software.

Before we can start using Matlab or Scilab, we must convert the data contained in the `.xsil` file into something that Matlab, Octave or Scilab can understand. To do this we use the utility program bundled with **xmds** called `xsil2graphics`.

To generate an input file for Matlab or Octave use either

```
% xsil2graphics lorenz.xsil
```

or

```
% xsil2graphics -matlab lorenz.xsil
```

however the second example is redundant as a Matlab or Octave `.m` file is the default output from `xsil2graphics`. You should see in the current directory a file called `lorenz.m` and a data file for the one moment group that we sampled for `lorenz1.dat`. For Scilab use

```
% xsil2graphics -scilab lorenz.xsil
```

giving the files `lorenz.sci` and `lorenz1.dat`. Ok, now we're ready to fire up our relevant numerical processing and graphical environment and visualise the results.

#### 1.1.7.1 Matlab and Octave

Start Matlab or Octave, and once at the command prompt load the information contained in the data file by using the command

```
>> lorenz
```

doing a `whos` should give you something similar to this

```
>> whos
  Name                Size                    Bytes  Class

  error_xOut_1        1x201                    1608  double array
```

```
  error_yOut_1          1x201                    1608  double array
  error_zOut_1          1x201                    1608  double array
  t_1                   1x201                    1608  double array
  xOut_1                1x201                    1608  double array
  yOut_1                1x201                    1608  double array
  zOut_1                1x201                    1608  double array

Grand total is 1407 elements using 11256 bytes
```

We can see that we've loaded our output data from the simulation into the variables `xOut_1`, `yOut_1` and `zOut_1` in Matlab or Octave. The reason why the `_1` is appended to the variable names is so that if one defines a variable in two different moment groups, but of the same name, then data isn't lost. The number refers to the label of the moment group in the simulation. At present you don't need to worry about these details, but just realise that they are there for when you write more complex scripts in the future. The error variables seen in the `whos` listing are the differences between the full-step integration and the half-step integration. The half-step integration is used for error checking purposes, so that you can check if your simulation is likely to be giving you reliable answers. Now plot the data by going

```
>> plot3(xOut_1, yOut_1, zOut_1)
>> xlabel('x')
>> ylabel('y')
>> zlabel('z')
```

and you should see a figure similar to Figure 1.1.

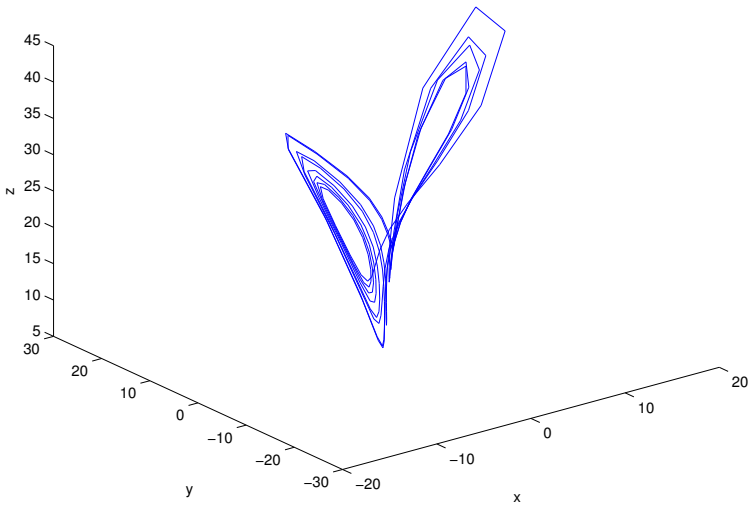



FIGURE 1.1: Three dimensional plot in Matlab of the trajectories of a Lorenz attractor. Parameters used were: $\sigma = 10$, $b = 8/3$, $r = 28$, with initial conditions of $x_0 = 1.0$, $y_0 = 1.0$, and $z_0 = 20.0$.

#### 1.1.7.2 Scilab

A very similar process is necessary for viewing the results in Scilab. Start up scilab, and at its command prompt run the command

```
-->exec('lorenz.sci')

-->temp_d1 = zeros(1,201);

-->t_1 = zeros(1,201);

-->xOut_1 = zeros(1,201);

-->yOut_1 = zeros(1,201);

-->zOut_1 = zeros(1,201);

-->error_xOut_1 = zeros(1,201);

-->error_yOut_1 = zeros(1,201);

-->error_zOut_1 = zeros(1,201);


-->lorenz1 = fscanfMat('lorenz1.dat');
Error Info buffer is too small (too many columns in your file ?)

-->temp_d1(:) = lorenz1(:,1);

-->xOut_1(:) = lorenz1(:,2);

-->yOut_1(:) = lorenz1(:,3);

-->zOut_1(:) = lorenz1(:,4);

-->error_xOut_1(:) = lorenz1(:,5);

-->error_yOut_1(:) = lorenz1(:,6);

-->error_zOut_1(:) = lorenz1(:,7);

-->t_1(:) = temp_d1(:);


-->clear lorenz1 temp_d1
```

to load the data into Scilab (you can safely ignore the warning), and then to obtain a graphical output of the data, run the command

```
-->param3d(xOut_1, yOut_1, zOut_1)
```

which should give something along the lines of that in Figure 1.2

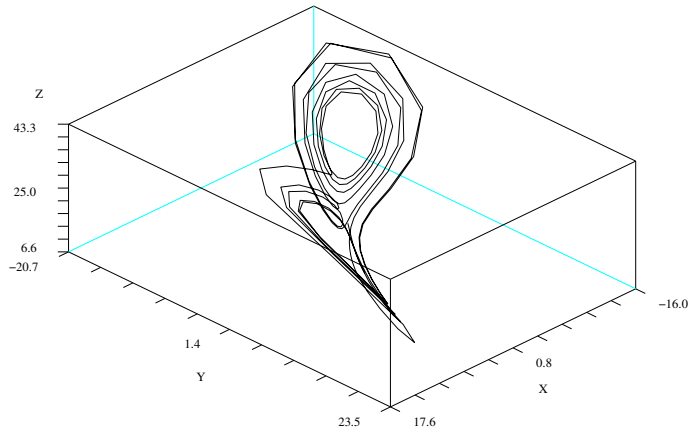



FIGURE 1.2: Three dimensional plot in Scilab of the trajectories of a Lorenz attractor. Parameters used were: $\sigma = 10$, $b = 8/3$, $r = 28$, with initial conditions of $x_0 = 1.0$, $y_0 = 1.0$, and $z_0 = 20.0$.

As we can see from both of these figures that we get the usual strange attractor "butterfly" shape. Now that we've spent a lot of time going over the very basics of writing a simulation from scratch, we now speed up a bit, and introduce some new **xmds** tags, but still with the theme that we are writing this all from a clean slate. Hopefully you will be able to see the other more powerful abilities of **xmds** and be able to start writing your own simulations.

## 1.2 A more complex simulation

In this section I'll introduce how to use Fourier space in your simulations (and the extra tags required), and explain why it is sometimes easier to perform part of the calculation in Fourier space and then transform back to position space. To illustrate these extensions to what we already know from Section 1.1 we'll look at solving the one-dimensional diffusion equation

$$\frac{\partial a(x,t)}{\partial t} = \kappa \frac{\partial^2 a(x,t)}{\partial x^2} \tag{1.4}$$

where $a(x,t)$ is the field to be evolved by the differential equation and is a function of time, $t$, and space $x$, and $\kappa$ is a constant describing how quickly the solution diffuses. An example of an application of the diffusion equation is for modelling the diffusion of (some initial distribution of) temperature in a metal rod; over time the temperature distribution will flow from areas of higher temperature to areas of lower temperature, eventually achieving a uniform distribution over the entire rod.

So why do we use Fourier space when solving this differential equation? The main reason is that it's a lot easier to calculate some of the differentials in Fourier space than it is in position space. It turns out that if one transforms position space into its respective Fourier domain, that a partial derivative with respect to position, just becomes $i$ (i.e. $\sqrt{-1}$) times the coordinate in Fourier space. For our example, such a mapping would be:

$$\frac{\partial}{\partial x} \mapsto ik_x. \tag{1.5}$$

Hence, in Fourier space, the second derivative on the right hand side of Equation (1.4) is just $-k_x^2$. Given that (discrete) Fourier transforms aren't that hard to do on a computer (and in **xmds** we use the Fastest Fourier Transforms in the West http://www.fftw.org, so they're pretty fast) using such a transformation improves the calculation somewhat.

### 1.2.1 Specifying the problem

We now need to specify the problem properly. To do this we must specify an initial condition for the solution we wish to evolve, and we must specify the boundary conditions of the domain over which we wish to solve this particular problem. Boundary conditions are necessary here since we have a transverse dimension (i.e. $x$) in this system (recall in Section 1.1 we had no transverse dimensions, only the propagation dimension of time). In following Garcia [1], as we do here again, we are trying to model the temperature diffusion of an initial temperature distribution in a one-dimensional rod, the ends of which are kept at a constant temperature of $T = 0$. Unfortunately, this implies Dirichlet boundary conditions, and **xmds** only implements periodic boundary conditions. This isn't strictly true, as one *can* implement absorbing boundary conditions in **xmds** (and people do in practice), however, one has to jump through some hoops that we don't really want to bother with here. So, to imitate Dirichlet boundary conditions, we'll not let the solution evolve outside the domain of the transverse dimension, $x$. All this means is that we make that particular domain rather larger than was necessary, and we make sure we don't evolve it in time for too long. We also start with a very narrow initial condition, and not have the diffusion coefficient too high, so as to inhibit the diffusion a bit. Please note that this is an example simulation to get people used to using the syntax of **xmds** and not necessarily to pedantically solve certain physical problems, we merely use the physical situations here to illustrate **xmds**, not the other way around.

An important solution of the diffusion equation is a Gaussian of the form [1]

$$T(x,t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \exp\left[\frac{-(x-x_0)^2}{2\sigma^2(t)}\right], \tag{1.6}$$

where $x_0$ is the location of the maximum and the standard deviation, $\sigma(t)$, increases with time as

$$\sigma(t) = \sqrt{2\kappa t}. \tag{1.7}$$

This solution is also a handy initial condition, and this is the analytical form of the initial condition we will be giving to **xmds**.

### 1.2.2 Starting off the simulation code

Now with the boundary conditions, and the initial condition specified analytically we are now in a position to start writing some **xmds** code. As per usual, we start with the `<?xml version="1.0"?>` and `<simulation>` tags. Then add the simulation name, which we'll call `diffusion`, and we'll put the author name and a brief description of what the simulation is supposed to do. The global variables we have in the problem we are solving are the diffusion coefficient $\kappa$, the standard deviation of the initial Gaussian distribution $\sigma$, and the positon of the mean of the Gaussian distribution $x_0$. These variables we'll call respectively, `kappa`, `sigma`, and `x0`. The code for this looks like:

```
<?xml version="1.0"?>
<simulation>

  <!-- Global system parameters and functionality -->
  <name> diffusion </name>
  <author> Paul Cochrane </author>
  <description>
    Solves the one-dimensional diffusion equation for an initial
    Gaussian pulse.  Adapted from A. L. Garcia, "Numerical Methods
    in Physics" (1994).
  </description>

  <!-- Global variables for the simulation -->
  <globals>
  <![CDATA[
      const double kappa = 0.1;
      const double sigma = 0.1;
      const double x0 = 0.0;
  ]]>
  </globals>

  <!-- more xmds code to come -->

</simulation>
```

which we put into a file called `diffusion.xmds`.

### 1.2.3 Describing the field

Again, the next thing to tell **xmds** about is the `<field>` element. This time we have a transverse dimension which is in the $x$ direction, and we mention this using the `<dimensions>` tag like so:

```
<dimensions> x </dimensions>
```

In the general case, **xmds** expects a space-separated list of transverse dimensions here, but in our example things are a bit simpler.

We now have to tell **xmds** the number of grid points of the lattice in this dimension, and over what domain in this dimension the grid is defined. To do these two things we use the `<lattice>` and `<domains>` tags. In our simulation here, we want to sample from $x = -1$ to $x = 1$, and use 100 points. Therefore, we set `<lattice>` to `100`, and `<domains>` to `(-1,1)`. Notice that the `<domains>` tag is defined by using an ordered pair syntax. **xmds** expects to see the domain for each transverse dimension defined as an ordered pair i.e. two comma-separated values enclosed in parentheses; if there is more than one transverse dimension, then the domains for each are defined by a list of space-separated ordered pairs. The last thing we need to mention before discussing the `<vector>` tag is that the number of samples is set to the same value as that in Section 1.1, i.e. 1, and that the name of the field, as per usual, is `main`. The `<field>` element now looks like

```
<field>
  <name> main </name>
  <dimensions> x </dimensions>
  <lattice> 100 </lattice>
  <domains> (-1,1) </domains>
  <samples> 1 </samples>

  <!-- more xmds code to come -->

</field>
```

We now need to specify the `<vector>` element. This is much the same as previously discussed in Section 1.1, but with some changes and one addition: the `<fourier_space>` tag. The vector `<name>` is again `main`, the `<type>` this time is `complex` though. The reason this might be confusing is because the temperature is a real quantity, and therefore, those who have been reading carefully may question why `float` wasn't chosen as the type instead. A type of `complex` is chosen because we are using a Fourier transform technique whereby the solution is transformed into Fourier space to calculate the second partial derivative in $x$ and then transformed back again, and to be able to transform into Fourier space, we need our variables to be of complex type. The `<components>` tag is set to `T`, since this is the name of the variable about to be defined in the `CDATA` block to come, and we tell **xmds** that this component is *not* defined in Fourier space by setting the `<fourier_space>` tag to `no`.

We next define the `CDATA` block. This is our C++ language representation of Equation (1.6), which we are using as our initial condition. The `CDATA` block is

```
<![CDATA[
    T = rcomplex(
          exp(-(x - x0)*(x - x0)/(2.0*sigma*sigma))
             /(sigma*sqrt(2.0*M_PI))
                 ,0.0);
]]>
```

This may look quite complicated, and possibly because we've tried to split the code over several lines in an attempt to break up the various parts and because this is intended for a fixed width page. The code does nevertheless introduce some important concepts. These are: the `rcomplex()` function, the ability to split lines of code over multiple lines if necessary,

and the M_PI variable

The rcomplex() function is one of a set of utility functions added as part of **xmds** to allow users to define complex variables. The syntax of rcomplex() is rcomplex(x,y) where x and y are real variables representing the real and imaginary parts of the complex number respectively. This explains one line of the CDATA block reads merely: ,0.0);. The lonely comma is just separating the two arguments to rcomplex(), the 0.0 is the imaginary part of the variable we are defining, which is real, but of complex type, and the closing parenthesis and semicolon just finish off the function call syntax.

Notice that the variable T is assigned to a quantity which is defined over multiple lines. Although this may seem strange to some, for those familiar with C language rules will know that all the C compiler is looking for is the semicolon as the character denoting the end of the expression. Therefore, it is possible to split equations over many lines to break up complicated expressions, or to highlight certain parts of the expression that may be important.

The M_PI variable is an automatically set variable by **xmds** that is the value of $\pi$, i.e. 3.14159...

Note also that one can use any of the standard mathematical functions defined in C/C++, such as sqrt(), log() etc.

This completes the discussion of the <field> element, which is now:

```
<!-- Field to be integrated -->
<field>
  <name> main </name>
  <dimensions> x </dimensions>
  <lattice> 100 </lattice>
  <domains> (-1,1) </domains>
  <samples> 1 </samples>

  <vector>
    <name> main </name>
    <type> complex </type>
    <components> T </components>
    <fourier_space> no </fourier_space>
    <![CDATA[
        T = rcomplex(
              exp(-(x - x0)*(x - x0)/(2.0*sigma*sigma))
                /(sigma*sqrt(2.0*M_PI))
                     ,0.0);
    ]]>
  </vector>
</field>
```

### 1.2.4 The sequence and integrate elements

Using what we know from Section 1.1, we now use <sequence> and <integrate> elements to describe the guts of what **xmds** has to do. We'll use here the RK4EX algorithm, an interval

of length 1, a lattice of 1000, and take 50 samples along the propagation direction.

Now, because we're evolving the solution partially in Fourier space, we need to define the operators that are going to be performing the evolution. This is done with the `<k_operators>` element. The reason why we call this the `<k_operators>` element is because Fourier space is often referred to as $k$-space, and position space as $x$-space, hence these operators are operating in $k$-space, and so they are $k$-operators. We next tell **xmds** that the $k$-operators (there is actually only one here) are constant over the course of the simulation by setting the `<constant>` tag to `yes`. We do this because if **xmds** has to assume that the $k$-operators *aren't* constant then it has to use much slower code to evolve the solution, and we are fortunate that for our simulation here the $k$-operators are constant since they can be calculated via the (also constant) `x` variable. **xmds** needs to know the name of the operator we are going to use for our $k$-operator, and this we set with the `<operator_names>` tag to be `L`. In general, the `<operator_names>` tag expects a space-separated list of the operator names you wish to define. We then give the C++ code necessary to define the operator in a `CDATA` block, and for our simulation this is

```
<![CDATA[
    L = -kappa*kx*kx;
]]>
```

Note that we have one variable here that we haven't defined before: `kx`. This is a variable automatically defined by **xmds** when we define $k$-operators. If we had another variable called `y` and defined a $k$-operator for it, then it would be called `ky`.

The last thing we need to do within this section of the script is tell **xmds** how to evolve the solution—in other words, the differential equation! As in Section 1.1 we use a `CDATA` block with a modified C++ syntax that **xmds** understands to write down the differential equation.

```
<![CDATA[
    dT_dt = L[T];
]]>
```

This completes the work necessary to integrate the solution forward, and completes the `<integrate>` and `<sequence>` elements, which are:

```
<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm>RK4EX</algorithm>
    <interval>1</interval>
    <lattice>1000</lattice>
    <samples>50</samples>
    <k_operators>
      <constant>yes</constant>
      <operator_names>L</operator_names>
      <![CDATA[
          L = -kappa*kx*kx;
      ]]>
    </k_operators>
```

```
    <![CDATA[
        dT_dt =  L[T];
    ]]>
  </integrate>
</sequence>
```

### 1.2.5 The output element

The <output> element is much like that discussed in Section 1.1. We need to specify a <filename> tag, just so that we are documenting everything nicely, and then <group> and <sampling> elements to describe the rest of the information **xmds** needs to properly sample the data and save it out to file. There are two new tags that are needed because we have a transverse dimension to worry about; these are <fourier_space> and <lattice>. The <fourier_space> tag is necessary to tell **xmds** if the sampling of the output is to be performed in Fourier space. **xmds** expects to see a space-separated list of yes or no values for each of the transverse dimensions, and for the situation here we don't want the output sampled in Fourier space and we only have one transverse dimension, so we set the <fourier_space> tag to no. The <lattice> tag tells **xmds** how finely the transverse dimensions should be sampled, and in general expects to see a space separated list of values telling it how many samples in the particular transverse dimension to take. Here we just set <lattice> to 50. The moments we are interested in sampling is of the temperature, so we specify a <moments> tag value of temperature and give the code to calculate this moment in a CDATA block as

```
<![CDATA[
    temperature = T;
]]>
```

All of this information gives the <output> element to be

```
<!-- The output to generate -->
<output>
  <filename>diffusion.xsil</filename>
  <group>
    <sampling>
      <fourier_space> no </fourier_space>
      <lattice>       50 </lattice>
      <moments>temperature</moments>
      <![CDATA[
          temperature = T;
      ]]>
    </sampling>
  </group>
</output>
```

### 1.2.6 The final script

We now have a complete script! And this is, in its entirety:

```
<?xml version="1.0"?>
<!-- Example simulation: Diffusion Equation -->

<simulation>

  <!-- Global system parameters and functionality -->
  <name>diffusion</name>
  <author> Paul Cochrane </author>
  <description>
    Solves the one-dimensional diffusion equation for an initial
    Gaussian pulse.  Adapted from A. L. Garcia, "Numerical Methods
    in Physics" (1994).
  </description>
  <prop_dim>t</prop_dim>

  <!-- Global variables for the simulation -->
  <globals>
  <![CDATA[
    const double kappa = 0.1;  // diffusion coefficient
    const double sigma = 0.1;  // std dev of initial Gaussian
    const double x0 = 0.0;     // mean position of initial Gaussian
  ]]>
  </globals>

  <!-- Field to be integrated over -->
  <field>
    <name>main</name>
    <dimensions>  x   </dimensions>
    <lattice>    100  </lattice>
    <domains>  (-1,1) </domains>

    <samples>1</samples>
    <vector>
      <name>main</name>
      <type>complex</type>
      <components>T</components>
      <fourier_space>no</fourier_space>
      <![CDATA[
        T = rcomplex(
              exp(-(x - x0)*(x - x0)/(2.0*sigma*sigma))/
                (sigma*sqrt(2.0*M_PI))
                     ,0.0);
      ]]>
    </vector>
```

```
  </field>

  <!-- The sequence of integrations to perform -->
  <sequence>
    <integrate>

      <algorithm>RK4EX</algorithm>
      <interval>1</interval>
      <lattice>1000</lattice>
      <samples>50</samples>
      <k_operators>
        <constant>yes</constant>
        <operator_names>L</operator_names>
        <![CDATA[
          L = -kappa*kx*kx;
        ]]>
      </k_operators>
      <![CDATA[
          dT_dt =  L[T];
      ]]>
    </integrate>
  </sequence>

  <!-- The output to generate -->
  <output>
    <filename>diffusion.xsil</filename>
    <group>
      <sampling>
        <fourier_space> no </fourier_space>
        <lattice>       50 </lattice>
        <moments>temperature</moments>
        <![CDATA[
          temperature = T;
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

Here is a link to the finished (gzipped) script file diffusion.xmds.gz on the **xmds** web site (http://www.xmds.org).

### 1.2.7 Making the simulation and getting results

Running through the sequence of events necessary to generate a simulation binary executable file, running it and producing the results for Matlab, Octave or Scilab, you should see a

sequence of events (and output) something like following:

Generating the simulation binary:

```
% xmds diffusion.xmds
```

Running the simulation:

```
% diffusion
Making forward plan
Making backward plan
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.000000e-02
<snip>
Sampled field (for moment group #1) at t        = 9.800000e-01
Sampled field (for moment group #1) at t        = 1.000000e-00
maximum step error in moment group 1 was 9.909189e-09
```

The forward and backward plans are the fftw routines calculating the necessary Fourier transforms.

Generating the Matlab or Octave output:

```
% xsil2graphics lorenz.xsil
Output file format defaulting to matlab.
Output file name defaulting to 'diffusion.m'
Proccessing xsil data container 1 ...
Writing data container 1 to file ...
```

Generating the Scilab output:

```
% xsil2graphics -scilab lorenz.xsil
Output file name defaulting to 'diffusion.sci'
Proccessing xsil data container 1 ...
Writing data container 1 to file ...
```

#### 1.2.7.1 Matlab

Loading the data into Matlab or Octave:

```
>> diffusion
```

Doing a `whos`:

```
  Name                      Size                    Bytes  Class

  error_temperature_1      50x51                20400  double array
  t_1                       1x51                  408  double array
  temperature_1            50x51                20400  double array
  x_1                       1x50                  400  double array

Grand total is 5201 elements using 41608 bytes
```

Plotting the data:

```
>> mesh(t_1, x_1, temperature_1)
>> xlabel('t')
>> ylabel('x')
>> zlabel('T')
```

you should see a figure similar to Figure 1.1.

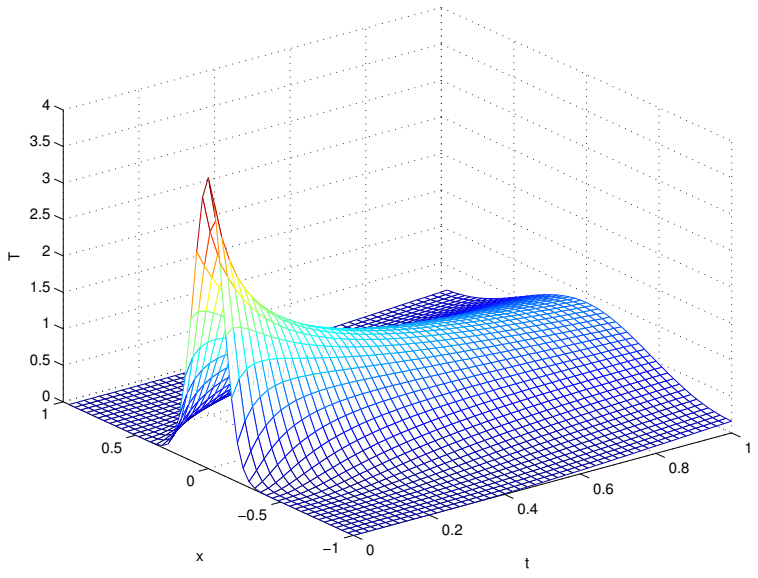



FIGURE 1.3: Three dimensional plot in Matlab of the diffusion of Gaussian pulse according to the diffusion equation. Parameters used were: $\kappa = 0.1$, $\sigma = 0.1$, $x_0 = 0$

#### 1.2.7.2 Scilab

Loading the data into Scilab:

```
-->exec('diffusion.sci')

-->temp_d1 = zeros(50,51);

-->t_1 = zeros(1,51);

-->temp_d2 = zeros(50,51);

-->x_1 = zeros(1,50);

-->temperature_1 = zeros(50,51);

-->error_temperature_1 = zeros(50,51);


-->diffusion1 = fscanfMat('diffusion1.dat');
Error Info buffer is too small (too many columns in your file ?)
```

```
-->temp_d1(:) = diffusion1(:,1);

-->temp_d2(:) = diffusion1(:,2);

-->temperature_1(:) = diffusion1(:,3);

-->error_temperature_1(:) = diffusion1(:,4);

-->t_1(:) = temp_d1(1,:);

-->x_1(:) = temp_d2(:,1);


-->clear diffusion1 temp_d1 temp_d2
```

Plotting the data:

```
-->plot3d(x_1, t_1, temperature_1)
```

should give something along the lines of that in Figure 1.2

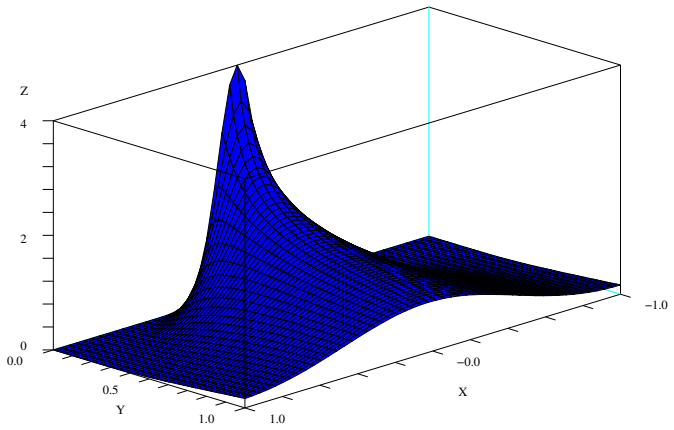



FIGURE 1.4: Three dimensional plot in Scilab of the diffusion of Gaussian pulse according to the diffusion equation. Parameters used were: $\kappa = 0.1$, $\sigma = 0.1$, $x_0 = 0$

Note that in both Figure 1.3 and Figure 1.4 we have an initial Gaussian pulse at $t = 0$, which then spreads out and loses amplitude as $t$ increases. This is the expected evolution of the solution according to the diffusion equation. The data points near the back of the graph, close to $t = 1$ are unlikely to be an accurate representation of the solution at this point, and indeed, are unlikely to be correct. Nevertheless, we have the expected behaviour, and have now demonstrated sufficient **xmds** tags for you, the user, to be confident to go off and write your own simulations. We wish you the very best of luck!

# 2 Extra and advanced features

## 2.1 Error checking

The error checking feature of **xmds** is enabled by default and is controlled by using the `<error_check>` tag. This is a boolean tag, and expects either a `yes` or `no` entry. In the context of **xmds**, error checking means to run the simulation twice: once through at the full step defined in the simulation script (via the `<lattice>` and `<interval>` assignments); and then again at half of the full step size. The maximum difference between the field values in each moment group is reported, and gives an indication of the discretisation error in the simulation. If one of the adaptive algorithms is chosen, error checking means that the simulation is run a second time with one 16th of the specified tolerance.

For instance, setting `<error_check>` to `yes` in the `atomlaser` simulation (this is an example code in the `examples` directory of the **xmds** distribution) we get the following output:

```
Making forward plan
Making backward plan
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
Beginning half step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
```

```
Sampled field (for moment group #1) at t        = 1.000000e-08
maximum step error in moment group 1 was 4.408207e-11
```

The error reported here is of the order of $10^{-11}$ and therefore we can be confident that discretisation error is not a significant problem in the simulation output. Once you are sure that your simulation is behaving nicely, you can then turn off error checking by setting `<error_check>` to `no` thereby speeding up the simulation. This is especially important for those whose simulations are going to take a *long* time to run. So, test your simulation to make sure that the error is low for a short simulation run, and then for the main run turn error checking off.

## 2.2 MPI: automatic parallelisation of simulations

One of the most powerful features of **xmds** is its ability to automatically parallelise simulations. We go through an extended discussion of this with specific focus on stochastic simulations in Chapter 4, but it is of worth mention here as well. Not only can **xmds** parallelise stochastic simulations by running each stochastic path on a separate computer, but it is also able to parallelise the computation of deterministic problems as well. It does this with the help of a package known as MPI, which stands for the Message Passing Interface, and is a means to organise the communication and computation associated with parallel simulations. To parallelise your simulation, all you have to do is add *one line of code!* Honestly. We're not joking. To turn this feature on in your code you need to add the line:

```
<use_mpi> yes </use_mpi>
```

and that's it. What **xmds** will do when running your simulation is split up the computation of the field and pass these parts of the overall computation to different computers, where it is solved faster than possible by doing so on a single machine. One good reason for splitting a simulation like this up and processing each part of the field on different processors is because for some very large simulations the memory requirements are too large, and therefore won't fit on one computer: using MPI is the only way to solve the problem.

There is one major caveat here however: *ONLY* use MPI for deterministic problems on a supercomputer, or a cluster setup where there is very small network latency (it doesn't take long for computers to talk to one another). This is very important, because the Fourier transforms require a lot of communication, and if the network between nodes of the cluster is slow then this will reduce the speed of the computation significantly, probably making it faster to run on a single cpu. Nevertheless, if you do have access to powerful computing facilities, then by all means, use this feature.

For stochastic problems, there is a second option you may wish to use, which changes the way the different paths are allocated between processors. This can be altered by using the `<MPI_Method>` tag, which can take the values "Scheduling" or "Uniform".

## 2.3 Benchmarking

To get an idea as to how long your code is going to run when you scale the various simulation parameters up for a long simulation, or just to see how long the main body of code takes, you can use the `<benchmark>` tag. This tag is a boolean which by default is `no`, but when set to `yes` it tells **xmds** to insert timing code around the main code block, excluding the fftw plan creation and deletion steps (this is because these steps are not in general indicative of how long the simulation will run, especially when scaled up to long simulation times). To use this feature just put the code

```
<benchmark> yes </benchmark>
```

into your simulation script in the global simulation and functionality section, namely before the `<globals>` tag.

The simulation will then report at the end of its run how long it took. An example of this is (again, using the `atomlaser` simulation):

```
Making forward plan
Making backward plan
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
Beginning half step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
maximum step error in moment group 1 was 4.408207e-11
Time elapsed for simulation is: 10 seconds
```

where the simulation has taken (approximately) 10 seconds to complete.

## 2.4 Wisdom

The `<use_wisdom>` tag is the way to enable FFTW's wisdom feature. This tag expects a boolean argument, and by default is set to `no`. However, when set to `yes` you can expect an immense increase in the startup speed of your simulations.

Wisdom is the name fftw gives to stored information about their Fourier transform plans. What fftw does before it decides to use a particular method for calculating the Fourier transform is to run some calculations beforehand to see which of the methods is the fastest (this can be related to your system's architecture, the size of the problem, etc.) and then it can optionally store this information so that fftw doesn't have to go through all of the

hard work again, and therefore make use of the stored "wisdom" about the problem at hand. Enabling wisdom means that subsequent runs of the simulation will start up and run (overall) much faster.

**xmds** requires a place to save this accumulated wisdom so that it can be reloaded in subsequent simulation runs. The way **xmds** does this is to save the widsom in a file called `<hostname>.wisdom`, where `<hostname>` is the name of the computer you are running the simulation on. Note that for simulations using MPI, that the `.wisdom` filename uses the format `<hostname><rank>.wisdom` where the `<rank>` is the MPI process rank number, and stops name conflicts when doing parallel simulations. There are two places that **xmds** can store `.wisdom` files: in the user's `~/.xmds/wisdom` directory; or in the directory local to the simulation. The former is used if the `~/.xmds/wisdom` directory exists, and the latter is used if not.

Running the `atomlaser` simulation with wisdom turned on, we get the following output (for the first run):

```
Performing fftw calculations
Making forward plan
Making backward plan
Keeping accumulated wisdom
Finished fftw calculations
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
Beginning half step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
maximum step error in moment group 1 was 3.802825e-11
Time elapsed for simulation is: 11 seconds
```

and then for the second run:

```
Performing fftw calculations
Standing upon the shoulders of giants... (Importing wisdom)
Making forward plan
Making backward plan
Keeping accumulated wisdom
Finished fftw calculations
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
```

```
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
Beginning half step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.500000e-09
Sampled field (for moment group #1) at t        = 5.000000e-09
Sampled field (for moment group #1) at t        = 7.500000e-09
Sampled field (for moment group #1) at t        = 1.000000e-08
maximum step error in moment group 1 was 3.802825e-11
Time elapsed for simulation is: 10 seconds
```

You will note, if you have run the `atomlaser` simulation both with and without wisdom how quickly the simulation starts once some fftw wisdom is used. Also, the simulation tells you that it is using previously generated wisdom, and that it is saving it for future use.

## 2.5 Binary output

When performing big simulations, i.e. over many dimensions or when propagating for a large distance over the propagation dimension, one is going to produce *very* large output files. This can be a problem, and the problem will be exacerbated by the fact that by default, **xmds** outputs data in ascii format, with a lot of redundancy. As a way to reduce the size of the output, **xmds** since `xmds-1.2` has had the ability to generate binary output files, which are inherently smaller (and can have better precision) than ascii data files, but also deals away with the redundancy introduced in the way that the ascii data is stored.

In `xmds-1.2` binary output was controlled by the `<binary_output>` and `<use_double>` tags. This syntax is deprecated as of `xmds-1.3` in favour of passing attributes to the `<output>` element, and this is the syntax we'll be discussing here. Users who are still using `xmds-1.2` are advised either to upgrade to a more recent version of **xmds** to make use of the better syntax. If you still wish to use `xmds-1.2`, then the syntax for using binary output is described in Chapter 11.

### 2.5.1 The `format` attribute

As mentioned above, the output format of an **xmds** simulation is now controlled by the `format` attribute of the `<output>` element. The syntax is as follows:

```
<output format="ascii"|"binary">
  <!-- other xmds tags -->
</output>
```

where by saying `"ascii"|"binary"` we mean that the format option is either the string `"ascii"` or `"binary"` (the double quotes are necessary) and that the default option is `"ascii"`.

For those who have used **xmds** in the past, you may remember that all of the data is output into the `.xsil` file. Binary output doesn't stop the generation of the `.xsil` file, but

merely uses a feature of the XSIL format that enables binary files to be pointed to by the `.xsil` file. Therefore, all of the important parameters of the simulation are still saved to the `.xsil` file, just the data is now saved to another file (or files if you have more than one moment group) containing just a binary string of data. So, when using binary output the following files will be produced: a `.xsil` file containing simulation parameters and pointing to the output data (by default, this will be called `<simulation name>.xsil`); and a binary data file for each moment group, being called in general `<simulation name>mg<moment group number>.dat`.

Running the `atomlaser` simulation with the `format` set to `"ascii"` we get an output `.xsil` file of size 808 kB. Now, if we run the `atomlaser` simulation again, except with the `<output>` tag set to

```
<output format="binary">
```

then we get a `.xsil` file of size 4 kB, and a `.dat` file called `atomlasermg0.dat` of size 336 kB, giving a total of 340 kB which is 42% smaller than with just ascii output. Bigger savings can be expected with longer simulations and/or simulations using more dimensions.

### 2.5.2 The `precision` attribute

The default binary output is at double precision. This is not always necessary for output of data, especially if the data is to be displayed graphically and then interpreted further there; the extra precision is not necessarily worthwhile. Therefore, there is also the `precision` attribute available in the `<output>` element, with which one can set the output precision to either single or double precision. The syntax for this is as follows:

```
<output format="binary" precision="double"|"single">
  <!-- more xmds tags -->
</output>
```

where `"double"|"single"` means the options are either `"double"` or `"single"` with `"double"` being the default option. Notice that the `format` attribute is also set to `"binary"` this is to emphasise that it is pointless specifying the `precision` without the `format` since the `precision` attribute is meaningless for ascii output.

Using this option, and rerunning the `atomlaser` simulation we find that the file size of `atomlasermg0.dat` is 168 kB and `atomlaser.xsil` is 4 kB, which overall is 21% of the size of the original ascii output.

## 2.6 Initialisation of field vectors from file

In Chapter 1, Section 1.1.3 we initialised the field inside the `<vector>` element by using C/C++ code. It is also possible to already have these vectors calculated and stored in a file, which **xmds** can then load and use to initialise the field. This feature can be useful if the calculation of the vectors is particularly difficult and you don't wish for **xmds** to have to calculate them, or you may have already generated the data from another program and so going through the hassle of getting **xmds** to recalculate the data is a waste of time. Anyway, it can be handy to do on some occasions and so **xmds** provides a means for you to do this via

the <filename> tag within the <vector> element within the <field> element. The syntax for this is:

```
<field>
  <vector>
    <filename format="ascii"|"binary"|"xsil">
      <!-- enter the file name here -->
    </filename>
  </vector>
<field>
```

where "ascii" is the default option when the format attribute is not specified.

As of xmds-1.3, **xmds** has the ability to load binary as well as ascii data. Which **xmds** should expect is given by the format attribute of the <filename> tag within the <vector> element. Using binary input, however, doesn't significantly change how the data should be organised prior to loading into an **xmds** simulation. If MPI is enabled **xmds** will only load into memory the appropriate part of the input file, irrespective of the file format.

### 2.6.1 Intialisation from an XSIL file

As of xmds-1.5-3, **xmds** can initialise a vector from a moment group of an XSIL file produced by a <breakpoint> tag (see Section 2.9) or an <output> tag in **xmds**. If you are generating the XSIL file from an <output> tag, then the output moment group must meet a certain format for **xmds** to be able to understand how to load the file correctly. If the file is generated from an <breakpoint> tag, then this is taken care of for you if the variables have the same names in the two simulations.

For XSIL files generated from output moment groups, the format of the XSIL file must be "binary" (not "ascii"). Also, the moment group number of the XSIL file that will be used for initialisation must be specified with the moment_group attribute of the <filename> tag if there is more than one moment group in the XSIL file, if there is only one moment group in the XSIL file (as is the case for XSIL files generated from a <breakpoint> tag), then this attribute can be omitted. If the vector is of type double, then the variables of the vector are initialised from the output moment group variables of the same name but suffixed with an 'R'. If the vector is of type complex, then the real and imaginary components of each variable are initialised by the values of the output moment group variables of the same name but with a suffix of 'R' for the real component and 'I' for the imaginary component. For example, the complex variables x and y would be initialised by the output moment group variables xR, xI, yR and yI, and you would use the following code in your <output> tag to create these variables:

```
<output>
        <group>
                <sampling>
                        <moments>xR xI yR yI</moments>
                        <![CDATA[
                                xR = x.re;
                                xI = x.im;
```

```
                                    yR = y.re;
                                    yI = y.im;
                                    ]]>
                </sampling>
        </group>
</output>
```

Not every variable in a vector need be present in the moment group of the XSIL file, as any variable that is not present is automatically initialised to zero, or by a CDATA section, as in Chapter 1, Section 1.1.3. Although **xmds** will continue initialisation even if it cannot find all the variables in a vector in the XSIL file (it will not continue if it cannot find any variables), it will print a warning about any variables that it cannot find in the XSIL file. Note that the sequence of initialisation steps for each element in a vector is to first initialise the element to zero, then to use any code in the CDATA section if present, and finally to initialise from the XSIL file if the variable is present in the moment group. Hence, initialisation from the XSIL file will override any initialisation in the CDATA section.

The dimensions of a vector can be initialised in any combination of $x$-space and $k$-space by using the `<fourier_space>` tag in the same way as it is used for initialisation from C/C++ code, however the default is that each dimension is initialised in $x$-space.

There are some restrictions on the geometry of the moment group in the XSIL file, however these conditions depend on whether the geometry matching mode (specified by the attribute `geometry_matching_mode` of the `<filename>` tag) is set to `"strict"` mode or `"loose"` mode. In `"strict"` mode, the following conditions apply:

1. The moment group must have the same number of dimensions as the field. In other words, the moment group can only have been sampled once, as sampling a moment group a number of times introduces an extra dimension, the propagation dimension.

2. The moment group's dimensions must have the same name and be in the same order as those of the field.

3. If a dimension is specified as being in $x$-space ($k$-space) in the moment group, then it must be initialised in $x$-space ($k$-space). This can be done using the `<fourier_space>` tag.

4. Each dimension of the initialisation moment group must have the same number of points as the corresponding dimension of the field, and the start and end coordinates must be the same as those for the initialisation moment group.

In other words, in `"strict"` mode, the geometry of the initialisation moment group must be the same (to within some small variation) as that of the field. Note that XSIL files generated by a `<breakpoint>` tag automatically satisfy conditions 1 and 2 if the dimensions of the two simulations are the same, and in the same order.

In the `"loose"` geometry matching mode, the last condition is relaxed to:

4. The step size in each dimension in the initialisation moment group must be the same as the step size in the corresponding dimension of the field.

5. Some of the moment group grid points must overlap (i.e. a vector with points at positions $x = 0, 2, 4, 6, 8$ cannot be initialised from a moment group with points at positions $x = 1, 3, 5, 7, 9$.) Note that points that aren't initialised by the moment group are set to zero, or can be initialised by the `CDATA` element if set.

The advantage of `"loose"` mode is that it allows one to break up a simulation into parts where each part requires a slightly different grid. For example, in the diffusion example in Chapter 1, Section 1.2, the restriction was made that the simulation is not evolved for long enough such that the field becomes non-zero at the edge of the grid. With `"loose"` mode, after running the simulation for some time on a small grid, if the state of the field is sampled at the end of the simulation, the simulation can be continued on a larger grid (though still keeping the same step size in that dimension, and ensuring that the grid points do overlap). Also, if one wishes to increase (or reduce) the number of points in a given dimension, and keep the width constant, initialise the state of the field with that dimension in $k$-space, as in this case, the requirement that the step size in the $k$-space dimension be the same is equivalent to the requirement that the width of that dimension in $x$-space remain the same. Hence, the number of points in $x$-space in that dimension can be increased (or reduced).

Note that a binary XSIL file produced on any architecture *can* be used on any other architecture (byte swapping is automatically done if the endianness of the machine running the simulation is different to the endianness of the XSIL file), and XSIL files with the output in single-precision can also be used.

In summary, the syntax for initialisation of a vector from an XSIL file is:

```
<field>
  <vector>
    <filename format="xsil" moment_group="N"
                geometry_matching_mode="strict"|"loose">
      <!-- enter the file name here -->
    </filename>
    <fourier_space> <!-- yes, no, ... --> </fourier_space>
    <![CDATA[
        // optional CDATA code
        ]]>
  </vector>
<field>
```

where `"strict"` mode is the default geometry matching mode.

### 2.6.2 Input data layout for ASCII and binary formats

We now know the syntax of how to tell **xmds** that we want to input data from file, we just now need to organise the data that we are going to input into the layout that **xmds** expects to see it. Let's see how this works by considering a simple example. Imagine we have three input vectors that we want to initialise with double precision data: `x`, `y` and `z`. Their values are:

```
x = [ -2.0 -1.0 0.0 1.0 2.0 ]
```

```
y = [ -5e-2 1e-3 -1e-5 2e-4 -7e-2 ]
z = [ 10 20 30 50 1e3 ]
```

We can see that they are all 5 elements long (this will equal the `<lattice>` assignment), and that they can contain numbers formatted in exponential notation. We'll save this data into a file called `input.dat`. **xmds** expects this data to be ordered in a particular way, which is related to the way the data is stored internally. This order is an interlacing of the elements of each vector, such that the first element of the first vector (in this case `x`) is expected as the first entry in the input file, then the first element of the second vector (in this case `y`) then the first element of the third vector (`z` here), and then the second element of the first vector and so on.

One way of describing this is in terms of C/C++ code. The data is expected in this format:

```
x[0]
y[0]
z[0]
x[1]
y[1]
z[1]
```

and so on until the end of the data. Another way of describing this is in terms of the actual data, and so here is how the file `input.dat` will look:

```
-2.0
-5e-2
10
-1.0
1e-3
20
0.0
-1e-5
30
1.0
2e-4
50
2.0
-7e-4
1e3
```

If this seems unnecessarily complicated—it is. However, this is the way the data is expected and so we have to behave the way **xmds** expects otherwise our simulation will not work properly. As it turns out, storing the data within memory in this fashion means that calculations are performed on contiguous blocks of memory, and therefore are a lot faster than if entire vectors were stored with their elements next to one another. This is a significant point for the memory utilisation internal to the simulation and for maintaining the speed of **xmds**

simulations. However, it may be possible in future versions of **xmds** for the input data to be specified more logically (i.e. have `x` defined first, then `y` etc.) and then for the simulation to reorganise the data internally so that calculations are performed efficiently and quickly.

If your input data is binary instead of ascii (as we have above), then you would use the `format="binary"` assignment in the `<filename>` tag, and then **xmds** would expect the data to be a string of double precison numbers in the same order as that given above.

### 2.6.3 Importing complex data

If we want to import complex data, we just specify the real then imaginary parts sequentially as pairs of data. Imagine that we now have two vectors (so we don't have to consider so many vectors) called `x` and `y`. They have values of (just for the sake of argument)

```
x = [ 1.2+2.0i 7.5+0.0i ]
y = [ -5e-2+10i 7e10-8e-7i ]
```

and they will be organised in the input file as follows:

```
real(x[0])
imag(x[0])
real(y[0])
imag(y[0])
real(x[1])
imag(x[1])
real(y[1])
imag(y[1])
```

which is

```
1.2
2.0
-5e-2
10
7.5
0.0
7e10
-8e-7
```

With complex data the binary input method is slightly different. The assignment to the `format` attribute is the same (i.e. `"binary"`), however, instead of separating the real and imaginary parts of the complex numbers that are to be read in, one just has the binary representation of the complex number to be read. So in a sense, the binary input of complex data is exactly the same as that of double data, except that the data is complex and not double (which seems obvious, but it sort of had to be said).

## 2.7 Command line arguments

Do you want to run your simulation many, many times ranging over several different global parameters? If the answer is yes, then the command line argument feature of **xmds** is for you. In versions of **xmds** before `xmds-1.2` to be able to map a parameter space, or run the program over many different values of a simple global variable, you had to modify your script, rerun **xmds** (with its implied compliation step) and then run the simulation *for each value*. This, put plainly sucked, so we put in a way to pass arguments to the simulation binary executable, enabling us to write a simple shell script (or Perl or Python) to run our simulation over many different values. This removed the need to recompile the simulation again and again, and generally speaking speeds things up and takes (at least some of) the pain out of doing things like mapping parameter spaces.

So, how do we tell **xmds** to make the simulation accept command line arguments? You do this with an `<argv>` tagset, which you put somewhere before the `<globals>` element. For those of you who have worked with C before and passing arguments to programs will notice that we've used the `argv` name here for the list of arguments the program will accept, in exactly the same way that C programming does by convention. To set up this list, we need to specify, the arguments, and the relevant properties of the arguments. As such we need to tell **xmds** what the name of the argument is, its data type, and its default value (for the instances when we don't want to specify the value on the command line). As might be obvious here, we have a nested structure of information, and hence the corresponding **xmds** code is similarly nested. The syntax of adding command line arguments to simulations is as follows:

```
<argv>
  <arg>
    <name> </name>                    <!-- the argument name -->
    <type> </type>                    <!-- data type of arg -->
    <default_value> </default_value> <!-- the default value -->
  </arg>
  <!-- more arg definitions here if necessary -->
<argv>
```

We'll now go through an example to show you how to use this feature, and some of the subtleties of using command line arguments with **xmds**-derived simulations. Let's revisit the `diffusion` simulation discussed in Chapter 1, Section 1.2. The main use of command line arguments is to be able to replace variables given in the `<globals>` element. Therefore, let's change the diffusion coefficient $\kappa$ (`kappa` in the code) to be an argument to the simulation. We do this by adding the following code before the `<globals>` element, and by commenting out the `kappa` declaration and assignment in the existing code. The **xmds** code then becomes:

```
<simulation>
  <!-- global parameters and functionality tags in here -->

  <!-- Command line arguments -->
  <argv>
    <arg>
```

```
      <name> kappa </name>
      <type> double </type>
      <default_value> 0.1 </default_value>
    </arg>
  </argv>

  <!-- Global variables for the simulation -->
  <globals>
  <![CDATA[
    // const double kappa = 0.1;  // diffusion coefficient
    const double sigma = 0.1;     // std dev of initial Gaussian
    const double x0 = 0.0;        // mean pos of initial Gaussian
  ]]>
  </globals>

  <!-- remainder of diffusion simulation xmds code -->
</simulation>
```

Notice that we've commented out the `kappa` variable using the C++ line comment style. This is just to remind us that `kappa` used to be there and is no longer, and what it was when we originally wrote the simulation. It can be a good idea to keep this kind of information around if you want, but it isn't necessary, and because it's a comment it will be ignored by the C/C++ compiler. Of course, if you *don't* comment the global declaration out, then the C/C++ compiler will throw an error and your simulation won't compile.

Running **xmds** on the file `diffusion.xmds` now gives a simulation binary that can accept arguments. You can try it out by running the simulation like so:

```
% diffusion --kappa 0.2
```

where we have run the `diffusion` simulation with `kappa` now set to 0.2.

**xmds** uses the GNU `getopt` set of functions to implement arguments, and as such supports both short and long option names. Therefore, the above example could have been run as

```
% diffusion -k 0.2
```

So, at the simplest level, **xmds** takes the long form of the argument name as the actual name of the variable, and takes the first character of the variable name for the short form of the argument. But what happens when you have two variables to be entered at the command line that start with the letter 'k'? What **xmds** does to solve this problem is, if a variable already has a short option taken (e.g. if we had already defined another variable in the `<argv>` list called say `kruntsch`), then the next character is used for the short option, which would be the letter 'a' for `kappa`. Of course, if this letter is taken then **xmds** searches for a single character representation of `kappa` throughout the variable name until it finds one that isn't used. If **xmds** doesn't find a short option that isn't used, then it throws an error.

Assuming that everything has worked ok, and the assignments to the short options have worked properly, how can one find out what the short option is if it has changed? Well, you can simply ask the simulation for help. Just run the simulation with either `-h` or `--help` and it will print out the usage of the simulation and a list of the option names, their data

type, and default value. For instance, asking the `diffusion` simulation for help we get the following output:

```
% diffusion --help
Usage: diffusion -k < double >

Details:
Option          Type            Default value
-k, --kappa     double          0.1
```

So, we call `diffusion` with `-k` and the simulation is expecting a double precision number after the `-k` flag. Also, we are told that either `-k` or `--kappa` are possible options (but we already knew that anyway), and that `kappa` is a double precision number of default value `0.1`.

And that's it! At present **xmds** can accept `int`, `double`, `float`, and `char *` for command line arguments. Complex numbers aren't yet implemented (as of `xmds-1.3`) but may be added in a future version.

Now, imagine that we wanted to run the `diffusion` simulation over a range of values starting from `0.1` to `1.0`. To do this we could write a simple shell script as follows:

```
#!/bin/sh

for i in 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
do
  echo "Running diffusion with kappa = $i"
  echo "i.e. diffusion --kappa $i"
  diffusion --kappa "$i"
  mv diffusion.xsil "diffusion_$i.xsil"
done
```

Notice that we've moved the output data file to a new filename since the file `diffusion.xsil` will be produced each time the simulation it is run, and hence our data it would get written over with new data each time the simulation is run, had we not bothered to rename the `.xsil` file.

Equivalently, we could have used a Perl script to do the same thing, for instance:

```
#!/usr/bin/perl -w

use strict;
my $i = 0.1;
while ($i <= 1.0) {
  print "Running diffusion with kappa = $i\n";
  print "i.e. diffusion --kappa $i\n";
  my @args = ("diffusion", "--kappa", $i);
  system(@args);
  `mv diffusion.xsil diffusion_$i.xsil`;
  $i = $i + 0.1;
}
```

Feel free now to extend (and have a play with) `diffusion.xmds` For example, change the

simulation to make it possible to vary `sigma` (and even `x0`) and see how the output changes, and in what regimes the assumption that our window size is large enough that the implicit periodic boundary conditions are unimportant (for a discussion of what this last part of the sentence means, have a look at Section 1.2.

## 2.8 Preferences

As of `xmds-1.3-1`, there has been the ability to have preferences specific to a given user. This gives the user more flexibility than before as they can control how simulations are built without having to recompile and reinstall **xmds**. Also, if **xmds** was installed by the root user, a non-root user of the system can modify how their simulations are built without having to reinstall the system binary and therefore alter how all other users' programs are built.

### 2.8.1 Turning preferences on and off

By default preferences are on, and are used if the preferences file can be found. If the preferences file cannot be found, or if `<use_prefs>` is set to `no` then the default settings defined at configuration and installation of **xmds** will be used. This is also the case for preference flags that are not specified in the preferences file: the default settings will be used. Therefore, one doesn't need to specify all of the settings able to be used, just the ones one wishes to change.

The `<use_prefs>` tag is set in the global configuration section of the **xmds** simulation script, namely at the beginning with the `<name>`, `<prop_dim>` etc. tags.

If `<use_prefs>` is set to `yes` explicitly (or not set at all) then **xmds** will search for a preferences file. This file is called `xmds.prefs` and can reside either in the user's `$HOME/.xmds` directory or the directory local to the simulation script being processed. **xmds** searches for the file in the user's home directory first and then in the current directory.

### 2.8.2 Setting preferences

Preferences are set in the `xmds.prefs` file by using key-value pairs delimited by an equals sign (`=`). Therefore, in general, the format is:

```
key = value
```

Spaces around the equals sign are ignored, such that the following are all equivalent:

```
key=value
key =value
key= value
key = value
```

The hash character (`#`) is used for comments; anything after and including the `#` are ignored when parsing the options. So, one can make what is happening much clearer what the flags are to do. For example:

```
# this is my funky preferences file
option = some_funky_value some_other_funky_value
other_option = also_quite_funky_variable  # isn't this variable funky?
```

the first line will be ignored and the text including and after the `#` on the third line will be ignored.

### 2.8.3 What are the options?

The options for building simulations that **xmds** accepts are:

- **xmds** options:

  `XMDS_CC` the C (C++) compiler **xmds** will use. Typical options include: `cc`, `gcc`, `g++`.

  `XMDS_CFLAGS` the flags passed to the C (C++) compiler. For those who are using the `make` utility, then these flags are the same as the `CFLAGS` variable ordinarily passed to the C compiler.

  `XMDS_LIBS` the libraries and library paths necessary to build the simulation. Again, for those familiar with `make` this is the same as the `LIBS` variable.

  `XMDS_INCLUDES` the include paths and files for the C (C++) compiler to look for when compiling the simulation.

- MPI options:

  `MPICC` the C (C++) compiler used by the local MPI implementation to compile the simulation. Often this is just `mpicc`, but on systems such as the APAC supercomputer in Canberra, Australia, this actually is just `cc`.

  `MPICCFLAGS` the `CFLAGS` variable to be passed to the MPI C (C++) compiler.

  `MPILIBS` the extra libraries necessary to compile a simulation for use with MPI. For instance, on cluster systems an MPI implementation such as LAM will be used. In this case, the extra libraries necessary to compile a simulation will be something like: `-lmpi -llam`.

- FFTW options:

  `FFTW_LIBS` the libraries and library paths specific to your fftw installation.

  `FFTW_MPI_LIBS` the libraries and library paths specific to your fftw installation necessary so that you can use fftw with MPI. Warning: only use fftw with a supercomputer. To perform Fourier transforms in parallel, a lot of communication between the nodes is necessary, hence this is only worthwhile on a supercomputer with a high speed network connection between nodes.

  `FFTW3_LIBS` the libraries and library paths specific to your installation of fftw version 3.

  `FFTW3_THREADLIBS` the additional libraries and library search paths required for using threaded fftw3 simulations.

- User defined options (at configuration):

  `USER_LIB` if **xmds** has been installed in the user's home directory, then this flag needs to be specified so that **xmds** can find the **xmds**-specific libraries when building a simulation.

  `USER_INCLUDE` if **xmds** has been installed in the user's home directory, then this flag needs to be specified so that **xmds** can find the **xmds**-specific header files when building a simulation.

### 2.8.4 Examples of changing options

**Using `gcc` for `g++`:** When configuring **xmds** before installation, the configuration script often sets the default C/C++ compiler for **xmds** to be `g++`. This is the GNU C++ compiler. Sometimes this is not desirable, and so one may wish to use the GNU C compiler (`gcc`) instead. To do this, one needs to change two things: the `XMDS_CC` setting, and the `XMDS_LIBS` setting. In the `xmds.prefs` file one then sets `XMDS_CC` to `gcc`, and appends `-lstdc++` to the list of flags already given for `XMDS_LIBS` at installation. The addition of `-lstdc++` is so that `gcc` can make use of the C++ extensions to `gcc` so that it can actually compile the simulation (which is in some sense a mix of C and C++).

**Using `icc` for `g++`:** An alternative C++ compiler to `g++` is the Intel C++ Compiler: `icc`. To use this compiler instead of `g++` or `gcc` (assuming of course that you have the compiler installed on your system) just set `XMDS_CC` to `icc` and prepend `-limf` to `XMDS_LIBS` (this adds the `icc` native support for its maths libraries).

**Debugging:** By default, **xmds** is configured to use quite aggressive optimisations when compiling simulations. If, however, you suspect something is going wrong and you wish to debug the simulation binary directly (using `gdb`, `dbx` or another symbolic debugger), then you will need to put symbolic debugging information into the binary executable. To do this, replace the default options by setting the `XMDS_CFLAGS` variable to `-g`.

**Profiling:** There may be instances when one wants to find out what part of the code is taking up the most time when running a simulation. This is generally speaking a part of debugging and testing a simulation and not normally part of using **xmds**. However, if you're interested in seeing what lines of code are using the most time, you'll want to add profiling information to the code. To do this you will need to add either the `-p` or `-gp` option to the `XMDS_CFLAGS` variable. The `-p` option generates extra code for profiling with the `prof` utility, and the `-gp` option generates extra code for profiling with the `gprof` utility.

## 2.9 Breakpoints

Breakpoint elements are parts of a simulation (similar to an `<integrate>` or a `<filter>` element) that cause the state of some vectors of the simulation to be saved to an XSIL file when the breakpoint element is hit. This can be used early in a simulation to enable you to check that, for example, the simulation isn't running off the grid, or that the behaviour is wrong and the simulation needs to be terminated. This way, much time can be saved waiting for the result of a long simulation that needs to be re-run anyway.

Another use of breakpoint elements is to save the state of some (or all) vectors to an XSIL file for loading by another simulation (as discussed earlier in Section 2.6.1 of this chapter). The naming convention for the vectors (and components of complex vectors) in the XSIL file produced is the same as that used for loading XSIL files as described in Section 2.6.1.

Although creating an XSIL file to be used for initialising another simulation can be achieved almost as easily with an output moment group, breakpoints should be used instead of output moment groups for large deterministic simulations that use MPI. Currently (`xmds-1.5-2`), because of the way in which output moment groups are sampled with MPI, each node allocates the entire memory required for sampling each output moment group. This means that sampling the entire field for a large simulation that will not fit into the memory of a single node is impossible, and hence creating an XSIL file from which the simulation could be continued is also impossible. As the intended use of moment groups is that they should be used to sample a small amount of data, an alternative solution was required for this situation. Breakpoint elements have been designed such that the additional memory use when used in a deterministic simulation with MPI is equal to the size of the field stored on any given node (and only while the XSIL file is being written), instead of the total field size (for the entire simulation).

The syntax for a breakpoint element (this should be in a `<sequence>` element) is:

```
<breakpoint>
        <filename> <!-- XSIL filename for output,
                        e.g. simulation.xsil--> </filename>
        <fourier_space> <!-- yes, no, ... -->  </fourier_space>
        <vectors> <!-- list of vectors to be saved to the file -->
        </vectors>
</breakpoint>
```

# 3 Using a template

In this tutorial, we're going to hack an **xmds** template to pieces to write a simulation script. This is possibly one of the easiest ways to make an **xmds** simulation script; almost everything has been done for you, and all that is left for you to do is to translate the equations into something a C/C++ compiler could understand. This tutorial builds upon the skills learnt in Chapter 1 and from the knowledge (of the existence of, at least) of the extra tags discussed in Chapter 2.

## 3.1 The advection equation

But first, the problem. Here we'll solve the one dimensional advection equation

$$\frac{\partial}{\partial t}A(x,t) = -v\frac{\partial}{\partial x}A(x,t), \tag{3.1}$$

where $A(x,t)$ is the field to evolve according to the differential equation, $x$ is the spatial dimension, $t$ is time, and $v$ is the velocity of the wave. As with the two examples discussed in Chapter 1, this example is adapted from that in Garcia [1].

We shall use in this example an initial pulse which is a cosine-modulated Gaussian:

$$A(x,t=0) = \cos[k(x-x_0)]\exp\left[-\frac{(x-x_0)^2}{2\sigma^2}\right], \tag{3.2}$$

where $k = 2\pi/\lambda$ is the wave number of the pulse, which has a wavelength $\lambda$, $x_0$ is the initial peak position of the pulse, and $\sigma$ is the initial pulse width (standard deviation). This initial

condition has an analytical solution (which we give here merely for interest value)

$$A(x,t) = \cos\{k[(x-vt)-x_0]\}\exp\left\{-\frac{[(x-vt)-x_0]^2}{2\sigma^2}\right\} \tag{3.3}$$

$$= \cos\{k[x-(x_0+vt)]\}\exp\left\{-\frac{[x-(x_0+vt)]^2}{2\sigma^2}\right\}, \tag{3.4}$$

which is still a cosine-modulated Gaussian, just displaced by an amount $vt$.

The boundary conditions we shall use are periodic, and since we get this for free when using **xmds**, there isn't anything special we need to do here, other than to keep in mind that the boundary conditions are periodic. As with the diffusion equation example, we'll make the $x$ domain go from -0.5 to 0.5, we'll set the value of $\sigma$ to 0.1, and $x_0$ to 0. The wave speed we'll set to 1. Any other parameters will be mentioned as we go through and hack with the code.

## 3.2 The template code

Here is the code that we're going to hack with. This is just an outline of many of the tags that we can use to perform a simulation using **xmds**. Most of them you should know by now, but one or two you may be unfamiliar with, this is ok, as we don't need them for what we want to do. Comments have been put all through the document to give a better idea of what the tags do and what their default value is (if any).

```
<?xml version="1.0"?>
<simulation>

  <!-- Global system parameters and functionality -->
  <name> </name>       <!-- the name of the simulation -->
  <author> </author>   <!-- the author of the simulation -->
  <description>
    <!-- a description of what the simulation is supposed to do -->
  </description>

  <prop_dim> </prop_dim>     <!-- name of main propagation dim -->

  <stochastic> no </stochastic>  <!-- defaults to no -->
  <!-- these three tags only necessary when stochastic is yes -->
  <paths> </paths>                <!-- no. of paths -->
  <seed> 1 2 </seed>              <!-- seeds for rand no. gen -->
  <noises> </noises>              <!-- no. of noises -->

  <use_mpi> no </use_mpi>             <!-- defaults to no -->
  <error_check> yes </error_check>    <!-- defaults to yes -->
  <use_wisdom> yes </use_wisdom>      <!-- defaults to no -->
  <benchmark> yes </benchmark>        <!-- defaults to no -->
  <use_prefs> yes </use_prefs>        <!-- defaults to yes -->
```

```
<!-- Global variables for the simulation -->
<globals>
<![CDATA[

]]>
</globals>

<!-- Field to be integrated over -->
<field>
  <name> main </name>
  <dimensions> </dimensions> <!-- transverse dims -->
  <lattice> </lattice>       <!-- no. of points for each dim -->
  <domains> (,) </domains>   <!-- domain of each dimension -->
  <samples> </samples>       <!-- sample 1st point of dim? -->

  <vector>
    <name> main </name>
    <type> complex </type>           <!-- data type of vector -->
    <components> </components>       <!-- names of components -->
    <fourier_space> </fourier_space> <!-- defined in k-space? -->
    <![CDATA[

    ]]>
  </vector>
</field>

<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm> </algorithm> <!-- RK4EX, RK4IP, ARK45EX, ARK45IP, SIEX, SI
    <iterations> </iterations> <!-- default=3 for SI- algs -->
    <interval> </interval>   <!-- how far in main dim? -->
    <lattice> </lattice>     <!-- no. points in main dim -->
    <samples> </samples> <!-- no. pts in output moment group -->

    <k_operators>
      <constant> yes </constant>         <!-- yes/no -->
      <operator_names> </operator_names>
      <![CDATA[

      ]]>
    </k_operators>

    <vectors> </vectors>     <!-- vector names -->
    <![CDATA[
```

```
      ]]>
    </integrate>
  </sequence>

  <!-- The output to generate -->
  <output format="binary" precision="single">
    <group>
      <sampling>
        <fourier_space> </fourier_space> <!-- sample in k-space? -->
        <lattice> </lattice>           <!-- no. points to sample -->
        <moments> </moments>           <!-- names of moments -->
        <![CDATA[

        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

## 3.3 Ripping it to bits

Now all we need to do is to work out what we need and don't need, and to throw the relevant numbers and equations into the relevant places. If the pace is too high, you might like to go back and re-read Chapter 1 to make sure you got it all.

### 3.3.1 Global system parameters and functionality

Looking at the block of code that sits between the start `<simulation>` tag and the `<globals>` element, we can add the following things: the name of the simulation is `advection`, the author is `Paul Cochrane`, and the description can be something like:

```
<description>
  Solves the one-dimensional advection equation for an initial
  cosine-modulated Gaussian pulse.
  Adapted from A. L. Garcia, "Numerical Methods in Physics" (1994).
</description>
```

The propagation dimension is in time, so we set `<prop_dim>` to `t`, the simulation *isn't* stochastic, so we can leave the code where it is, or remove it since it's set to the default setting anyway. We'll remove it in this case as we don't gain anything, and it's extra code floating around that we don't need. Since this isn't a stochastic simulation, this implies that the `<paths>`, `<seed>` and `<noises>` tags are unnecessary. Also, we don't need the `<use_mpi>` tag either, because we've not met it before, the simulation is not stochastic, so making it use MPI (the message passing interface for parallel simulations) is a waste of time, and the default setting is off, so again, we rip this tag out.

We might as well leave error checking on (it's the default anyway), and leave the <error_check> tag the way it is. This is especially important at the early stages of writing a simulation, as one can then get an indication of the discretisation error of the simulation and this information can tell us how to tweak the simulation parameters if necessary. Leaving the <error_check> tag in the code is a good idea, as we'll probably want to switch it off at some later stage, and if we leave it in the code we can remember to do this if we want.

We definitely want to use FFTW's wisdom feature. This is because we have a differential in space, and it is nice and easy to define the operator in $k$-space, implying that we'll need to use Fourier transforms, and using "wisdom" speeds up the startup time of our simulations. Hence, we leave <use_wisdom> as is.

Benchmarking the code doesn't take up much room in the script, nor does it take up much time in the code, and it can be interesting to have around, so we'll leave the <benchmark> tag there. However, preferences are on by default anyway, and switching them off probably won't be of much use, so we'll get rid of the <use_prefs> tag, which will set it to the default value of yes.

We've seen in Chapter 2 how one can use command line arguments with the <argv> tagset. At this stage of simulation writing it's a good idea just to get things going, and not worry about varying these parameters just yet, so what we'll do is set them in the <globals> block and add the <argv> stuff in later if we want to.

The first chunk of code is now:

```
<!-- Global system parameters and functionality -->
<name> advection </name>      <!-- the name of the simulation -->
<author> Paul Cochrane </author>  <!-- the author of sim -->
<description>
  Solves the one-dimensional advection equation for an initial
  cosine-modulated Gaussian pulse.
  Adapted from A. L. Garcia, "Numerical Methods in Physics"(1994).
</description>

<prop_dim> t </prop_dim>    <!-- name of main propagation dim -->

<error_check> yes </error_check>   <!-- defaults to yes -->
<use_wisdom> yes </use_wisdom>     <!-- defaults to no -->
<benchmark> yes </benchmark>       <!-- defaults to no -->
```

### 3.3.2 Global variables for the simulation

Our global variables are $v$, $k$, $x_0$ and $\sigma$ (from inspection of the equations in Section 3.1), therefore, we need to use the following code for the <globals> block.

```
<!-- Global variables for the simulation -->
<globals>
<![CDATA[
    const double v = 1.0;
    const double x0 = 0.0;
    const double sigma = 0.1;
```

```
    double k = M_PI/sigma;
]]>
</globals>
```

Notice that we've not used the `const` keyword in front of the `k` declaration and assignment. This is because we're deriving the value of the wave number from the standard deviation of the wave, and so the value itself (as far as C/C++ is concerned) isn't a constant. If you're worried that this may be a problem further down the track, don't, because it isn't.

### 3.3.3 The field to be integrated over

The field, as per normal, has a name of `main`. We have one dimension other than the propagation dimension, and that is `x`, which we want to put say 50 grid points (i.e. `<lattice>` is set to 50), and as mentioned earlier, we want the domain to go from -0.5 to 0.5, so the `<domains>` tag is set to `(-0.5,0.5)`. We want to sample the first point of this dimension, so we set `<samples>` to `1`.

Within the `<vector>` assignments we only have one vector to define, and since we have to call at least one vector `main`, we'll use that. It will have to have a complex type because we're going to be using Fourier space for part of the integration, hence we set `<type>` to `complex`. There is only one component we have to define, and that is the field `A` that we'll be integrating over. We therefore use `A` as the `<components>` assignment. This isn't going to be defined in Fourier space, so the `<fourier_space>` assignment is set to `no`.

The trickiest part here is now defining the equation in the `CDATA` block. Recalling the initial condition in Equation (3.2), we therefore declare the variable `A` as

```
A = rcomplex(
      cos(k*(x-x0)) * exp(-(x-x0)*(x-x0)/(2.0*sigma*sigma)), 0.0);
```

where we have used the `rcomplex` function with a zero imaginary argument to define the (initially real) quantity, however, of `complex` type.

The result of these definitions gives us the following code for the `<field>` element:

```
  <!-- Field to be integrated over -->
  <field>
    <name> main </name>
    <dimensions> x </dimensions> <!-- transverse dims -->
    <lattice> 50 </lattice>       <!-- no. pts for each dim -->
    <domains> (-0.5,0.5) </domains> <!-- domain of each dim -->
    <samples> 1 </samples>        <!-- sample 1st point of dim? -->

    <vector>
      <name> main </name>
      <type> complex </type>          <!-- data type of vector -->
      <components> A </components>    <!-- names of components -->
      <fourier_space> no </fourier_space> <!-- def in k-space? -->
      <![CDATA[
          A = rcomplex(
              cos(k*(x-x0)) * exp(-(x-x0)*(x-x0)/(2.0*sigma*sigma))
```

```
                , 0.0);
    ]]>
   </vector>
  </field>
```

### 3.3.4 The sequence of integrations to perform

Now we come to the <sequence> section of the code. There's only one such sequence in this simulation, so we really only need to worry about the <integrate> block. The algorithm we'll use here is the fourth-order Runge-Kutta method in the explicit picture. We want the wave to circle the system the once, so we'll choose the <interval> and <lattice> so that it does so, while having a step size of about 0.002. We therefore set <interval> to 1, and <lattice> to 500. Only 50 points are really necessary for the output of this, hence <samples> is set to 50.

The next section to define is the <k_operators> element. The $k$-space operator is constant in time, and we'll call it L (which is sort of a convention in the **xmds** community). Since the operator acts in Fourier space, the spatial derivative merely becomes a multiplication, and so we can set the CDATA block (within the <k_operators> element) to:

```
<![CDATA[
    L = rcomplex(0.0, -v*kx);
]]>
```

Note that we have used the rcomplex() function, and that the operator is complex. This is because of the mapping of derivatives in $x$-space to $k$ vectors in $k$-space; remember that

$$\frac{\partial}{\partial x} \mapsto ik_x. \tag{3.5}$$

There are no <vectors> to define, so we can just delete this line, however, we must write down the differential equation we're trying to solve. So, remembering Equation (3.1), and the fact that **xmds** uses a special syntax for writing the differential equation in the <integrate> element, the CDATA section within the <integrate> element is:

```
<![CDATA[
    dA_dt = L[A];
]]>
```

Note that we *didn't* write down the equations as

```
<![CDATA[
    dA_dt = -v*L[A];
]]>
```

and define the operator as L = rcomplex(0.0, kx);. This is because of the way **xmds** transplants the equations into the code, and specifying the constants within the operators is the most general way **xmds** can do this. Heed a warning here though: **xmds** at present doesn't pick this kind of error up, so, one can write the equation as just mentioned and **xmds** will happily do its thing, however, your answers will be *wrong*. So, the advice here is to be *really careful*.

The <sequence> element now looks like this:

```
<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm> RK4EX </algorithm> <!--RK4EX,RK4IP,ARK45EX,ARK45IP,SIEX,SIIP
    <interval> 1 </interval>   <!-- how far in main dim? -->
    <lattice> 500 </lattice>     <!-- no. points in main dim -->
    <samples> 50 </samples><!-- no. pts in output moment group-->

    <k_operators>
      <constant> yes </constant>          <!-- yes/no -->
      <operator_names> L </operator_names>
      <![CDATA[
          L = rcomplex(0.0, -v*kx);
      ]]>
    </k_operators>

    <![CDATA[
        dA_dt = L[A];
    ]]>
  </integrate>
</sequence>
```

### 3.3.5 The output to generate

We're in the home stretch now! All we need to do is tell **xmds** what to spit out at the end of the simulation. We'll use ascii output (a bit more portable) and remove the precision assignment; we won't sample the output moments in Fourier space; we only have one dimension to sample (namely `x`) and we'll use 50 points here; there is only one moment to output, and that is the field amplitude which we'll call `amp`; and finally the code to sample the output moment is simply equal to the amplitude of the field which is just `A`, so the code put into the `CDATA` block is `amp = A;`. The `<output>` element code is therefore:

```
<!-- The output to generate -->
<output format="binary" precision="single">
  <group>
    <sampling>
      <fourier_space> no </fourier_space><!--sample in k-space?-->
      <lattice> 50 </lattice>           <!-- no. pts to sample -->
      <moments> amp </moments>          <!-- names of moments -->
      <![CDATA[
          amp = A;
      ]]>
    </sampling>
  </group>
</output>
```

### 3.3.6 The final program

And that's it! The finished simulation is this:

```
<!-- Global system parameters and functionality -->
<name> advection </name>      <!-- the name of the simulation -->
<author> Paul Cochrane </author>  <!-- the author of sim -->
<description>
  Solves the one-dimensional advection equation for an initial
  cosine-modulated Gaussian pulse.
  Adapted from A. L. Garcia, "Numerical Methods in Physics"(1994).
</description>

<prop_dim> t </prop_dim>    <!-- name of main propagation dim -->

<error_check> yes </error_check>   <!-- defaults to yes -->
<use_wisdom> yes </use_wisdom>     <!-- defaults to no -->
<benchmark> yes </benchmark>       <!-- defaults to no -->

<!-- Global variables for the simulation -->
<globals>
<![CDATA[
    const double v = 1.0;
    const double x0 = 0.0;
    const double sigma = 0.1;
    double k = M_PI/sigma;
]]>
</globals>

<!-- Field to be integrated over -->
<field>
  <name> main </name>
  <dimensions> x </dimensions> <!-- transverse dims -->
  <lattice> 50 </lattice>       <!-- no. pts for each dim -->
  <domains> (-0.5,0.5) </domains> <!-- domain of each dim -->
  <samples> 1 </samples>       <!-- sample 1st point of dim? -->

  <vector>
    <name> main </name>
    <type> complex </type>          <!-- data type of vector -->
    <components> A </components>    <!-- names of components -->
    <fourier_space> no </fourier_space> <!-- def in k-space? -->
    <![CDATA[
        A = rcomplex(
            cos(k*(x-x0)) * exp(-(x-x0)*(x-x0)/(2.0*sigma*sigma))
              , 0.0);
    ]]>
```

```
    </vector>
  </field>

  <!-- The sequence of integrations to perform -->
  <sequence>
    <integrate>
      <algorithm> RK4EX </algorithm> <!--RK4EX,RK4IP,ARK45EX,ARK45IP,SIEX,SIIP-
      <interval> 1 </interval>   <!-- how far in main dim? -->
      <lattice> 500 </lattice>     <!-- no. points in main dim -->
      <samples> 50 </samples><!-- no. pts in output moment group -->

      <k_operators>
        <constant> yes </constant>          <!-- yes/no -->
        <operator_names> L </operator_names>
        <![CDATA[
            L = rcomplex(0.0, -v*kx);
        ]]>
      </k_operators>

      <![CDATA[
          dA_dt = L[A];
      ]]>
    </integrate>
  </sequence>

  <!-- The output to generate -->
  <output format="ascii">
    <group>
      <sampling>
        <fourier_space> no </fourier_space><!--sample in k-space?-->
        <lattice> 50 </lattice>           <!-- no. pts to sample-->
        <moments> amp </moments>          <!-- names of moments -->
        <![CDATA[
            amp = A;
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

Here is a link to the finished (gzipped) script file advection.xmds.gz on the **xmds** web site (http://www.xmds.org).

## 3.4 Making the simulation and getting results

As per usual, we just need to run **xmds** on the simulation script, and then run the simulation. So, here we go:

```
% xmds advection.xmds
Output file name defaulting to 'advection.xsil'
compiling ...
        g++ -pthread  -O3 -ffast-math -funroll-all-loops
        -fomit-frame-pointer -o advection advection.cc
        -I/home/cochrane/bin -lstdc++ -lm  -lxmds
        -L/home/cochrane/bin -lfftw_threads -lfftw
advection ready to execute
```

then:

```
% advection
Performing fftw calculations
Standing upon the shoulders of giants... (Importing wisdom)
Making forward plan
Making backward plan
Keeping accumulated wisdom
Finished fftw calculations
Beginning full step integration ...
Sampled field (for moment group #1) at t        = 0.000000e+00
Sampled field (for moment group #1) at t        = 2.000000e-02
Sampled field (for moment group #1) at t        = 4.000000e-02
<snip>
Sampled field (for moment group #1) at t        = 9.600000e-01
Sampled field (for moment group #1) at t        = 9.800000e-01
Sampled field (for moment group #1) at t        = 1.000000e-00
maximum step error in moment group 1 was 7.801465e-06
Time elapsed for simulation is: 0 seconds
```

That was fast eh? And the error isn't too bad at about $10^{-5}$, so we can be vaguely confident of the results. Lets look at them now, in both Matlab (or Octave) and Scilab.

### 3.4.1 Matlab and Octave

Using `xsil2graphics` we generate the Matlab or Octave script by the command:

```
% xsil2graphics advection.xsil
```

and then running the following commands in Matlab or Octave:

```
>> advection
>> mesh(t_1,x_1,amp_1)
>> xlabel('t')
>> ylabel('x')
>> zlabel('A')
```

Doing all this should produce something very similar to that in Figure 3.1.

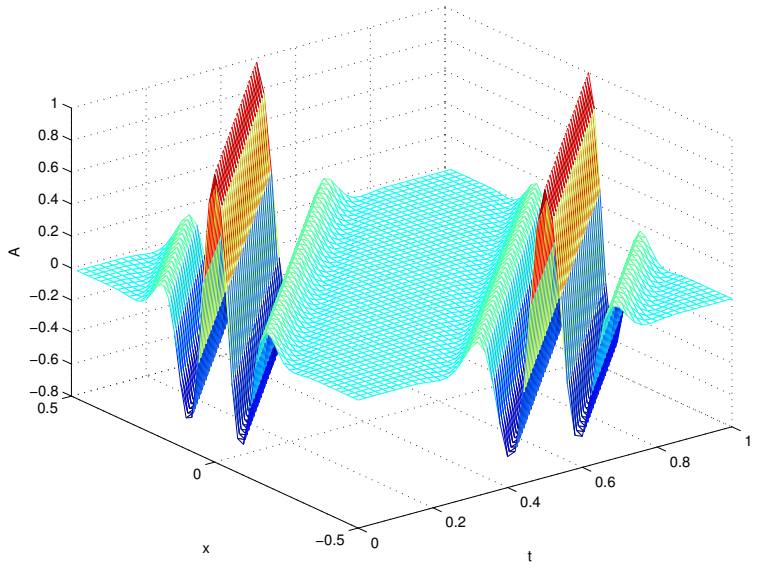



FIGURE 3.1: Three dimensional plot in Matlab of a cosine-modulated Gaussian pulse according to the advection equation. Parameters used were: $v = 1$, $\sigma = 0.1$, $x_0 = 0$, $k = \pi/\sigma$. Notice that with the periodic boundary conditions that the pulse moves off one side of the figure and re-enters from the opposite side.

### 3.4.2 Scilab

Using `xsil2graphics` we generate the Scilab script by the command:

```
% xsil2graphics -scilab advection.xsil
```

and then running the following commands in Scilab:

```
-->exec('advection.sci')

-->temp_d1 = zeros(50,51);

-->t_1 = zeros(1,51);

-->temp_d2 = zeros(50,51);

-->x_1 = zeros(1,50);

-->amp_1 = zeros(50,51);

-->error_amp_1 = zeros(50,51);


-->advection1 = fscanfMat('advection1.dat');
Error Info buffer is too small (too many columns in your file ?)

-->temp_d1(:) = advection1(:,1);
```

```
-->temp_d2(:) = advection1(:,2);

-->amp_1(:) = advection1(:,3);

-->error_amp_1(:) = advection1(:,4);

-->t_1(:) = temp_d1(1,:);

-->x_1(:) = temp_d2(:,1);


-->clear advection1 temp_d1 temp_d2



-->plot3d(x_1,t_1,amp_1)
```

which should generate something similar to that in Figure 3.2.

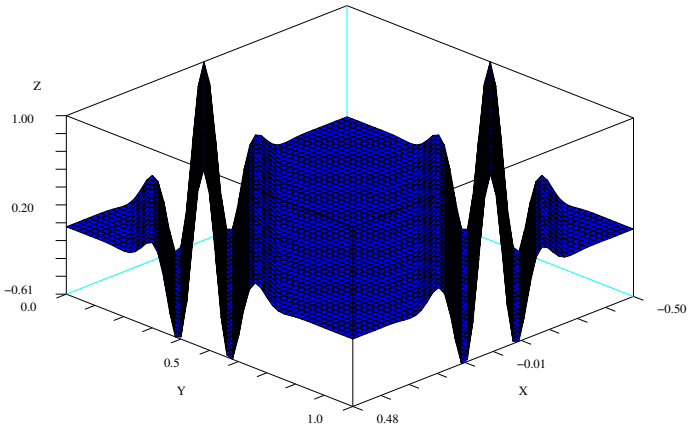



Figure 3.2: Three dimensional plot in Scilab of a cosine-modulated Gaussian pulse according to the advection equation. Parameters used were: $v = 1$, $\sigma = 0.1$, $x_0 = 0$, $k = \pi/\sigma$. Notice that with the periodic boundary conditions that the pulse moves off one side of the figure and re-enters from the opposite side.

## 3.5 Adding command line arguments

Now that we're happy with how the simulation is performing, we can define some command line arguments so that we can investigate the system more easily. We might as well be able to change all of the global variables except for $k$ (since it depends on one of the other variables), so add the following code snippet just before the `<globals>` block:

```
<!-- Command line arguments -->
<argv>
  <arg>
    <name> v </name>
    <type> double </type>
    <default_value> 1.0 </default_value>
  </arg>
  <arg>
    <name> x0 </name>
    <type> double </type>
    <default_value> 0.0 </default_value>
  </arg>
  <arg>
    <name> sigma </name>
    <type> double </type>
    <default_value> 0.1 </default_value>
  </arg>
</argv>
```

But remember to comment out these variables in the `<globals>` block otherwise the compiler will throw an error.

Now you can have a play! Try different variable options and see how the equations and **xmds** perform.

## 3.6 The xmds –template option

A new feature of **xmds**, as of version 1.3-3, is the output of a template code. When **xmds** is called with the `-template` (or equivalently the `-t`) option, then a template code will be written to either standard output (i.e. the terminal), or to file, if a filename is given after the `--template` flag. For instance, if one entered the following at the command line:

```
% xmds --template
```

then you would see several lines of **xmds** code scroll past. What's the use in it just scrolling past you say? Well, you could pipe this output to file like so:

```
% xmds --template > new_xmds_file.xmds
```

Nice eh? However, it is possible to save on keystrokes by getting **xmds** to make the file directly, saving you from having to use shell-related commands to save the output. To save a template directly to file, just enter this command:

```
% xmds --template new_xmds_file.xmds
```

and this will make a new file for you called `new_xmds_file.xmds` in the same directory as **xmds** was called. Why have the ability to just send the template to the screen? Well, doing so doesn't slow things down, and it gives an extra level of flexibility, and we're here to make your life as the user of **xmds** as easy as possible.

# 4 Stochastic simulations and MPI

One of the most powerful features of **xmds** is its ability to automatically parallelise stochastic (funky way to say "random") simulations. To illustrate this power we're going to have a look at an example of a stochastic differential equation (SDE) both with and without the use of MPI (the Message Passing Interface used for running parallel simulations). The physical model we'll be investigating is the simplified Kubo oscillator model taken from Gardiner [3].

$$\frac{dz}{dt} = i[\omega + \sqrt{2\gamma}\xi(t)]z \tag{4.1}$$

This system is a model of an oscillator with a mean frequency $\omega$ perturbed by a noise term $\xi(t)$. The parameter $\gamma$ describes the strength of the noise perturbation. Examples of where this could be used to model an actual physical system are in the theories of magnetic resonance and laser physics, and in single molecule spectroscopy.

## 4.1 Without MPI

First off, let's solve this problem without MPI. For those of you who don't know, MPI is the Message Passing Interface and is the current de-facto standard for performing computer simulations in parallel. MPI is a very powerful library of routines that allow one to perform different parts of a simulation concurrently on different processors and be able to handle the relevant communication between processors, or to run the same simulation with different parameters on different processors, thereby in both situations speeding up the solution of the given problem. Other similar systems exist, such as PVM (the Parallel Virtual Machine) but MPI is effectively an extension of this system and is very well developed and works well on supercomputers and cluster systems. MPI is also what **xmds** uses to parallelise its simulations so it gives us further motivation to only discuss it here.

We now take our template, and hack around with it a bit. Our propagation dimension

is time, so we set <prop_dim> to t. This time, however, we are performing a stochastic simulation, so we need to keep the <stochastic>, <paths>, <seed> and <noises> tags. Setting <stochastic> to yes, the number of paths to 1024, leaving the <seed> setting as-is, and noting that we only have one noise term, we get the following chunk of code describing the stochastic part of our simulation:

```
<stochastic> yes </stochastic>  <!-- defaults to no -->
<!-- these three tags only necessary when stochastic is yes -->
<paths> 1024 </paths>                <!-- no. of paths -->
<seed> 1 2 </seed>             <!-- seeds for rand no. gen -->
<noises> 1 </noises>             <!-- no. of noises -->
```

We'll leave the <use_mpi> tag where it is at present, similarly with the rest of the tags down to the <globals> section. We've got three variables in this simulation: omega, the mean frequency; gam, the perturbation strength [1]; and zo, the initial value of z [2]. The <globals> block comes out to this:

```
<!-- Global variables for the simulation -->
<globals>
<![CDATA[
  double omega = 0;    // mean frequency
  // the word gamma is already used in many maths libraries,
  // hence we use gam here
  double gam = 0.1;    // perturbation strength
  double zo = 1;       // initial value of z
]]>
</globals>
```

There are no transverse dimensions, so we can get rid of the <dimensions>, <lattice> and <domains> tags since they are all required just for the transverse dimensions. We want to sample the first point, so we set <samples> to 1. Our variable z is complex so we leave the first two tags of the <vector> block as is, and set the <components> tag to z, which we don't define in Fourier space, and we just set to the initial value, zo. This gives a <field> block of

```
<!-- Field to be integrated over -->
<field>
  <name> main </name>
  <samples> 1 </samples>        <!-- sample 1st point of dim? -->

  <vector>
    <name> main </name>
```

[1] Notice that we don't actually use the word gamma here. This is because gamma is often used in maths libraries for such things as the gamma function etc, so we try and avoid name conflicts as much as possible, and use the word gam.

[2] we could have just set z to some number later in the setup of the field, however, this way allows us to change the initial value of z later without having to dig through too much code, and we can easily see how to replace the <globals> block with an <argv> block and so have more dynamic control over the variables being put into the simulation.

```
    <type> complex </type>          <!-- data type of vector -->
    <components> z </components>    <!-- names of components -->
    <fourier_space> no </fourier_space> <!--defined in k-space?-->
    <![CDATA[
      z = zo;
    ]]>
  </vector>
</field>
```

In the `<sequence>` block we set the algorithm to `SIEX` because a semi-implicit algorithm does a better job of integrating stochastic equations than the fourth-order Runge-Kutta. We'll explicitly set the number of iterations to the default of `3` (we might want to change it in the future), we'll integrate for 10 seconds so set `<interval>` to `10`. We'll use 1000 for the lattice size and take 100 samples of it in the output moment group. There aren't any $k$-operators here, so we can delete that entire section (and the `<vectors>` tag). The interesting bit is where we write in the differential equation to be solved, which is really easy to do and is written in the `CDATA` block as

```
    <![CDATA[
      dz_dt = i*omega*z + i*sqrt(2.0*gam)*n_1*z;
    ]]>
```

where we've expanded out the brackets of the Kubo oscillator equation Equation (4.1). Notice that the value of $xi$ mentioned in Equation (4.1) has been replaced by the variable `n_1`. This is because when we tell **xmds** that we have noises it automatically creates some noise variables for us. In the present case we only have one noise, so there is only one variable, namely `n_1`. However, in the general case, where we could have say $m$ noises, we would have the noise variables `n_1`, `n_2`, ..., `n_m`.

Putting these pieces together gives a `<sequence>` block that looks like this

```
<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm> SIEX </algorithm> <!--RK4EX, RK4IP, SIEX, SIIP-->
    <iterations> 3 </iterations>  <!-- default=3 for SI- algs -->
    <interval> 10 </interval>     <!-- how far in main dim? -->
    <lattice> 1000 </lattice>     <!-- no. points in main dim -->
    <samples>100</samples> <!--no. pts in output moment group-->

    <![CDATA[
      dz_dt = i*omega*z + i*sqrt(2.0*gam)*n_1*z;
    ]]>
  </integrate>
</sequence>
```

To generate the `<output>` block we choose the format to be `ascii`, and remove the (now superfluous) precision attribute. The sampling isn't done in Fourier space so we can set that to `no`, we want to sample on a lattice of 100 points, and take the moments `realz` and `imagz`, which are the real and imaginary parts of `z` respectively. These are then defined in

the CDATA block by the following code

```
        <![CDATA[
          realz = real(z);
          imagz = imag(z);
        ]]>
```

We therefore have an <output> block of

```
  <!-- The output to generate -->
  <output format="ascii">
    <group>
      <sampling>
        <fourier_space>no</fourier_space> <!--sample in k-space?-->
        <lattice> 100 </lattice>      <!-- no. points to sample -->
        <moments> realz imagz </moments>  <!-- names of moments -->
        <![CDATA[
          realz = real(z);
          imagz = imag(z);
        ]]>
      </sampling>
    </group>
  </output>
```

and a final output simulation of

```
<?xml version="1.0"?>
<simulation>

  <name>kubo_tutorial</name>      <!-- the name of the sim -->

  <author> Paul Cochrane </author>  <!-- the author of the sim -->
  <description>
    Kubo oscillator example simulation.  This formally represents a
    simple model of an oscillator with a mean frequency omega
    perturbed by a noise term xi(t).
    Adapted from the example given in "Handbook of Stochastic
    Methods", C. W. Gardiner (1997)
  </description>

  <!-- Global system parameters and functionality -->
  <prop_dim> t </prop_dim>    <!-- name of main propagation dim -->

  <stochastic> yes </stochastic>  <!-- defaults to no -->
  <!-- these three tags only necessary when stochastic is yes -->
  <paths> 1024 </paths>               <!-- no. of paths -->
  <seed> 1 2 </seed>             <!-- seeds for rand no. gen -->
  <noises> 1 </noises>              <!-- no. of noises -->

  <use_mpi> no </use_mpi>            <!-- defaults to no -->
```

```
<error_check> yes </error_check>   <!-- defaults to yes -->
<use_wisdom> yes </use_wisdom>     <!-- defaults to no -->
<benchmark> yes </benchmark>       <!-- defaults to no -->
<use_prefs> yes </use_prefs>       <!-- defaults to yes -->

<!-- Global variables for the simulation -->
<globals>
<![CDATA[
  double omega = 0;   // mean frequency
  // the word gamma is already used in many maths libraries,
  // hence we use gam here
  double gam = 0.1;   // perturbation strength
  double zo = 1;      // initial value of z
]]>
</globals>

<!-- Field to be integrated over -->
<field>
  <name> main </name>
  <samples> 1 </samples>       <!-- sample 1st point of dim? -->

  <vector>
    <name> main </name>
    <type> complex </type>         <!-- data type of vector -->
    <components> z </components>  <!-- names of components -->
    <fourier_space> no </fourier_space> <!--defined in k-space?-->
    <![CDATA[
      z = zo;
    ]]>
  </vector>
</field>

<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm> SIEX </algorithm> <!--RK4EX, RK4IP, SIEX, SIIP-->
    <iterations> 3 </iterations>  <!-- default=3 for SI- algs -->
    <interval> 10 </interval>     <!-- how far in main dim? -->
    <lattice> 1000 </lattice>     <!-- no. points in main dim -->
    <samples>100</samples> <!--no. pts in output moment group-->

    <![CDATA[
      dz_dt = i*omega*z + i*sqrt(2.0*gam)*n_1*z;
    ]]>
  </integrate>
</sequence>
```

```
  <!-- The output to generate -->
  <output format="ascii">
    <group>
      <sampling>
        <fourier_space>no</fourier_space> <!--sample in k-space?-->
        <lattice> 100 </lattice>      <!-- no. points to sample -->
        <moments> realz imagz </moments>  <!-- names of moments -->
        <![CDATA[
          realz = real(z);
          imagz = imag(z);
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

The gzipped script code can be downloaded from the **xmds** web site http://www.xmds.org via the link: kubo_tutorial.xmds.gz

### 4.1.1 Making the simulation and getting results

You should be pretty good at doing this now, so we'll just show you the output of the various operations and not explain much.

```
xmds kubo_tutorial.xmds
Output file name defaulting to 'kubo_tutorial.xsil'
compiling ...
        g++ -pthread  -O3 -ffast-math -funroll-all-loops
        -fomit-frame-pointer -o kubo_tutorial kubo_tutorial.cc
        -I/home/cochrane/bin -lstdc++ -lm  -lxmds -L/home/cochrane/bin
        -lfftw_threads -lfftw
kubo_tutorial ready to execute
```

then

```
% kubo_tutorial
Beginning full step paths
Starting path 1
Starting path 2
<snip>
Starting path 1023
Starting path 1024
maximum step error in moment group 1 means was 1.010483e-04
Time elapsed for simulation is: 5 seconds
```

That was pretty good for doing 1024 paths twice at different time steps eh? Admittedly we're not pushing things much here. We'll do so soon though.

#### 4.1.1.1 Matlab and Octave

Using `xsil2graphics` we generate the Matlab or Octave script by the command:

```
% xsil2graphics kubo_tutorial.xsil
```

Running the following sequence of commands in Matlab or Octave

```
>> kubo_tutorial
>> plot(t_1,mean_realz_1)
>> xlabel('t')
>> ylabel('z')
```

generates Figure 4.1.

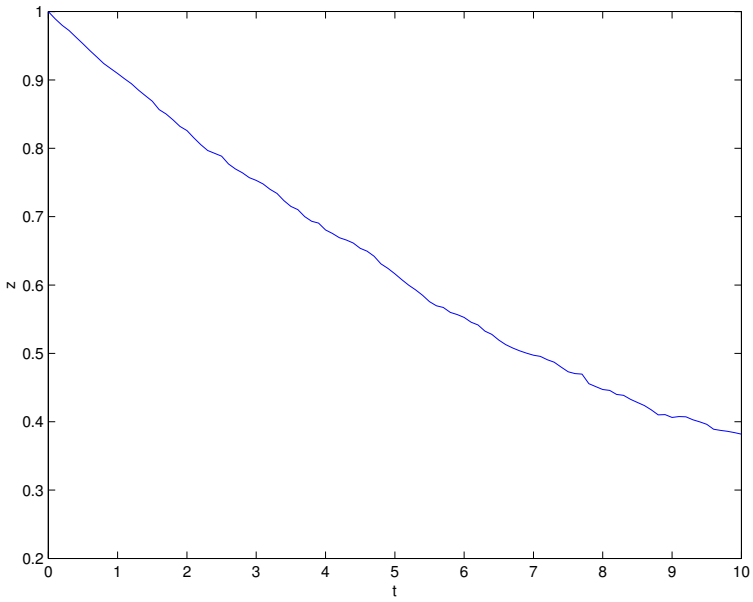



Figure 4.1: Matlab generated plot of the evolution of the oscillator frequency $z$ over time perturbed by noise.

#### 4.1.1.2 Scilab

Using `xsil2graphics` we generate the Scilab script by the command:

```
% xsil2graphics -scilab kubo_tutorial.xsil
```

Running the following sequence of commands in Scilab

```
-->exec('kubo_tutorial.sci')

-->temp_d1 = zeros(1,101);

-->t_1 = zeros(1,101);

-->mean_realz_1 = zeros(1,101);
```

```
-->mean_imagz_1 = zeros(1,101);

-->sd_realz_1 = zeros(1,101);

-->sd_imagz_1 = zeros(1,101);

-->error_realz_1 = zeros(1,101);

-->error_imagz_1 = zeros(1,101);


-->kubo_tutorial1 = fscanfMat('kubo_tutorial1.dat');
Error Info buffer is too small (too many columns in your file ?)

-->temp_d1(:) = kubo_tutorial1(:,1);

-->mean_realz_1(:) = kubo_tutorial1(:,2);

-->mean_imagz_1(:) = kubo_tutorial1(:,3);

-->sd_realz_1(:) = kubo_tutorial1(:,4);

-->sd_imagz_1(:) = kubo_tutorial1(:,5);

-->error_realz_1(:) = kubo_tutorial1(:,6);

-->error_imagz_1(:) = kubo_tutorial1(:,7);

-->t_1(:) = temp_d1(:);


-->clear kubo_tutorial1 temp_d1
```

generates Figure 4.2.

Notice that the results we get here are similar to the other Kubo oscillator example mentioned both later in this document and on the **xmds** web site. This is intentional, as we set $\omega = 0$ which should give the same answer, however, this simulation is more general than the later one discussed in Chapter 10. The main difference careful readers will have noticed is that the average solution of $z$ does not decay away to zero. This is because we've used a perturbation constant of 0.1 as opposed to $\frac{1}{\sqrt{2}}$ used in the later example.

### 4.1.2 Making the simulation hard

Now's the time to up the ante. In some real life situations, lots and lots of paths are required so that decent statistics are generated. For instance, in modelling the evolution of stock prices one can use a stochastic differential equation, however, for banks etc to be *really*

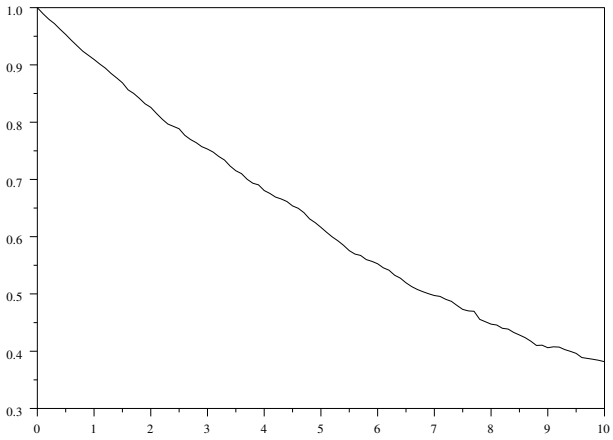



Figure 4.2: Scilab generated plot of the evolution of the oscillator frequency $z$ over time perturbed by noise.

sure of the results, they need to run of the order of millions of paths. This can take a *very* long time, especially if run on a single processor. Therefore, in such situations, being able to automatically parallelise ones simulation and run the different paths over many CPUs would be great, and this is something that **xmds** does really well, as you shall (hopefully) see.

Let's run the simulation over a few more paths, so we'll just glibly throw a some zeroes at the `<paths>` tag and set it to 1024000, and run the simulation again.

```
% kubo_tutorial
Beginning full step paths
Starting path 1
Starting path 2
<snip>
Starting path 1023999
Starting path 1024000
maximum step error in moment group 1 means was 9.161489e-05
Time elapsed for simulation is: 1479 seconds
```

Hmmm, that took a bit longer didn't it? That's just over 24 minutes. Surely we can do better than that! This is where we need to use many computers to solve the problem and so, this is where MPI comes in.

## 4.2 With MPI

Now we hope that by farming each path off to a different CPU that we can get an improvement in speed. Let's try it and find out.

At this stage we should expect that since each individual path doesn't take very long to run then the main overhead is going to be communication between farming a job off and sending the results back. In a more complex simulation it would be normal (and in fact

much better) for each node to spend quite some time working on an individual path before sending the information back; such a scenario is far more efficient computational use of CPU time. In any case, we should see a shorter wall time (the amount of time taken as far as the clock on the wall is concerned) for the job to complete.

To implement the use of MPI, make sure that your copy of **xmds** is built with MPI enabled. To do this one merely needs to run the configure script with the `--enable-mpi` option specified. From that point, the only thing you need to change within your XMDS script to switch back and forth between MPI and non-MPI code is a single flag at the top. Set the `<use_mpi>` tag to read `yes` or `no` as desired. How to run the resulting program will depend on the specific implementations of MPI that you are using. We will show a typical case based on the LAM/MPI implementation (another major implementation is MPICH), but details will vary from system to system.

### 4.2.1 Example using LAM/MPI

One of the first things to do is to make a hosts file so that LAM can know what machines it needs to send jobs to. The hosts file is just a list of hostnames of computers you have access to, that can run MPI. For example here is a sample hosts file called `lamhosts`

```
% cat lamhosts
bec00
bec01
bec02
bec03
bec04
bec05
bec06
bec07
bec08
bec09
bec10
bec11
bec12
bec13
bec14
bec15
```

As you can see there are 16 computers available to run jobs on

Next it's a good idea to do a `recon` to check that all of your hosts are working and they can accept MPI connections. To do this, run the following command, and you should see similar output.

```
% recon -v lamhosts
recon: -- testing n0 (bec00)
recon: -- testing n1 (bec01)
recon: -- testing n2 (bec02)
recon: -- testing n3 (bec03)
recon: -- testing n4 (bec04)
```

```
recon: -- testing n5 (bec05)
recon: -- testing n6 (bec06)
recon: -- testing n7 (bec07)
recon: -- testing n8 (bec08)
recon: -- testing n9 (bec09)
recon: -- testing n10 (bec10)
recon: -- testing n11 (bec11)
recon: -- testing n12 (bec12)
recon: -- testing n13 (bec13)
recon: -- testing n14 (bec14)
recon: -- testing n15 (bec15)
-----------------------------------------------------------------------------
Woo hoo!

recon has completed successfully.  This means that you will most likely
be able to boot LAM successfully with the "lamboot" command (but this
is not a guarantee).  See the lamboot(1) manual page for more
information on the lamboot command.

If you have problems booting LAM (with lamboot) even though recon
worked successfully, enable the "-d" option to lamboot to examine each
step of lamboot and see what fails.  Most situations where recon
succeeds and lamboot fails have to do with the hboot(1) command (that
lamboot invokes on each host in the hostfile).
-----------------------------------------------------------------------------
```

If everything has run successfully, then you can start MPI on each of the nodes (other computers) by using the `lamboot` command. This gets each of the other computers ready to get input from a parallel job, and doesn't actually start doing any computation. You should see output similar to this

```
% lamboot -v lamhosts

LAM 6.5.6/MPI 2 C++/ROMIO - University of Notre Dame

Executing hboot on n0 (bec00 - 1 CPU)...
Executing hboot on n1 (bec01 - 1 CPU)...
Executing hboot on n2 (bec02 - 1 CPU)...
Executing hboot on n3 (bec03 - 1 CPU)...
Executing hboot on n4 (bec04 - 1 CPU)...
Executing hboot on n5 (bec05 - 1 CPU)...
Executing hboot on n6 (bec06 - 1 CPU)...
Executing hboot on n7 (bec07 - 1 CPU)...
Executing hboot on n8 (bec08 - 1 CPU)...
Executing hboot on n9 (bec09 - 1 CPU)...
Executing hboot on n10 (bec10 - 1 CPU)...
Executing hboot on n11 (bec11 - 1 CPU)...
Executing hboot on n12 (bec12 - 1 CPU)...
```

```
Executing hboot on n13 (bec13 - 1 CPU)...
Executing hboot on n14 (bec14 - 1 CPU)...
Executing hboot on n15 (bec15 - 1 CPU)...
topology done
```

Now MPI is set up, we need change *one line* of code to make our simulation parallel. Change the `<use_mpi>` tag to read `yes` and you're done! Ok, now run **xmds** over the script to rebuild the binary executable, giving you something akin to the following:

```
% xmds kubo_tutorial.xmds
Output file name defaulting to 'kubo_tutorial.xsil'
compiling for MPI parallel execution ...
        mpicc -pthread  -O3 -ffast-math -funroll-all-loops
        -fomit-frame-pointer -lm  -lmpi -llam   -lstdc++
        -I/home/cochrane/bin -L/home/cochrane/bin -o kubo_tutorial
        kubo_tutorial.cc -I/home/cochrane/bin -I/usr/local//include/
        -lfftw_threads  -lfftw -L/usr/local//lib/
kubo_tutorial ready to execute
```

We now run the simulation by using the command:

```
% mpirun -c 16 kubo_tutorial
```

The command line flag `-c` to the `mpirun` program tells `mpirun` how many processes to generate over the machines we have, and since we have 16 computers to choose from, we set this to 16 here (it could be higher if you want to run more than one process per CPU).

Running this simulation took almost exactly 6 minutes. Well, that wasn't a great speed improvement was it? Especially given that we used 16 computers instead of 1, and now we've got a wall time improvement of about a factor of 4. As I mentioned above, some of this will be due to communication overhead i.e. too much of the computer's CPU is used in just sending the data back and forth across the network, and not enough time is actually spent just doing the calculation. However, we have reduced the amount of time necessary to come to a solution, so we have actually done what we set out to do.

A new option for the MPI routines within **xmds** is to implement load balancing. This is where the work of a parallel simulation is allocated to each of the CPUs running part of the job on the basis of the load of the CPU; CPUs with less load on average get more work to do, and CPUs with more load on average have less work to do. The adaptive scheduler that performs this operation can be switched on by the use of the code

```
<MPI_Method>Scheduling</MPI_Method>
```

or disabled (which may be useful if your system doesn't allow threads, for example, by the code:

```
<MPI_Method>Uniform</MPI_Method>
```

After you've finished your simulation, and generated the results, all that is left to do is to clean up after having run a parallel simulation. You do this by running the `wipe` program:

```
% wipe -v lamhosts
```

And that's it. We've managed to decrease the amount of wall time necessary to calculate many paths of a stochastic simulation, and all it took was for us to change one line of code!

# Part II

# Numerical Modelling Theory

# 5 Introduction

Unfortunately, appropriate mathematical models for the majority of natural phenomenon generate equations that are *non-integrable*. This means that the solutions to the problem cannot be written as exact analytic expressions, and those who would seek solutions must appeal to the modern computer to calculate them by a numerical procedure. The capability of modern computers continues to grow at a phenomenal rate, and hence so also does the field of computational physics. In fact it has been described by some as a "third way of doing physics", in addition to the more standard theoretical and experimental approaches. Computational physics represents an unusual marriage of these two approaches (which traditionally have always been rivals) in that the governing equations must be based on believable theory, yet each simulation is very much like an experiment to determine what actually happens. However, due perhaps to its rapid growth, computational physics is in need of improvement in at least one particular area. In a recent review titled *Microscopic simulations in physics* [4] Ceperley writes:

*Sadly, the lore of experimental and theoretical physics has not yet fully penetrated into computational physics. Before the field can advance, certain standards, which are commonplace in other technical areas, need to be adopted so that people and codes can work together.*

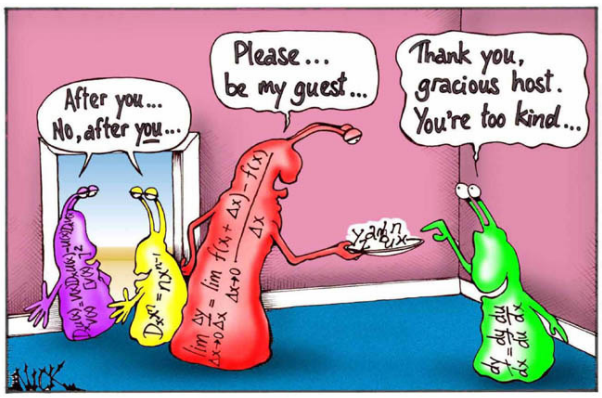
Here Ceperley is addressing the issue of standards and reproducibility within computational physics. The results of one group must be able to be independently verified by another group, and should one group cease working in a particular area it should be not unnecessarily difficult for another group to pick up where they left off. However, code writing is inherently individualistic and does not naturally lend itself to these properties. Further, the range of problems that encompass every physical phenomenon known to our species is not small, and if every problem had to be individually coded into a detailed computer program, checked, and debugged—well, the future would look bleak.

What is needed is to be able to specify the generalities of the problem without having to continually reprogram the detailed mechanisms of solving it. This approach is known as *high*

*level* programming, and there are a variety of both commercial and public license packages based on this approach. Unfortunately, to achieve the necessary flexibility these packages tend to be "interpreters", and thus they are seldom efficient at calculating the solution. And for some problems, those that take fast computers days and weeks to solve, efficiency is paramount. The only way to obtain efficiency is to directly compile a well written *low level* program, letting the compiler do the optimisation – which is what compilers are very good at. What many mathematicians and scientists would wish to have is their own personal expert computer programmer, who takes their high level description of the problem and writes a low level program specifically dedicated to solving it. Further, it is essential that this process is *error free*.

Presented here is a solution to these difficulties. **xmds** is a computer program that *generates* computer programs. **xmds** interprets a high level description of a problem, and in turn writes a low level computer program that a compiler can compile and optimise. The executable file produced then solves the problem as quickly and efficiently as possible. Further, once **xmds** is debugged, the process of code generation, at least from the high level script onwards, becomes error free. Of course, **xmds** cannot be written to be able to process absolutely any problem; **xmds** has its own scope of capability, which is outlined in Chapter 8.

We begin by outlining some of the basic theory involved in numerical modelling, and then follow on with the develpoment strategy behind **xmds** and the structure of the source code. We then look at the functionality of **xmds**, and illustrate this with worked examples. We close with a brief outlook for future development.




Deferential equations.

# 6 Numerical Modelling Theory

## 6.1 Differential Equations

The majority of numerical models for real world problems are based on one or more differential equations involving the various parameters concerned. The large majority of which may be written in the very general form:

$$\left[A\left(x,y\right)\frac{\partial^2}{\partial x^2}+2B\left(x,y\right)\frac{\partial^2}{\partial x\partial y}+C\left(x,y\right)\frac{\partial^2}{\partial y^2}\right]a^i\left(x,y\right)=f^i\left(x,y,\mathbf{a},\frac{\partial a^i}{\partial x},\frac{\partial a^i}{\partial y}\right). \tag{6.1}$$

These come under three sub-classifications [5] according the the parameters $A$, $B$, and $C$:

1. The PDE is said to be *elliptic* in regions of the $x$-$y$ plane where $AC-B^2>0$.

2. The PDE is said to be *parabolic* in regions of the $x$-$y$ plane where $AC-B^2=0$.

3. The PDE is said to be *hyperbolic* in regions of the $x$-$y$ plane where $AC-B^2<0$.

This model for classifying PDEs is only two dimensional, but many problems in physics exhibit up to four dimensions. However, the majority of PDEs in physics are able to be expressed in a form where one of the dimensions is special; special in that there are only ever first order derivatives involving that dimension. A two dimensional form of such a PDE looks like:

$$C\left(x,y\right)\frac{\partial^2}{\partial y^2}a^i\left(x,y\right)=f^i\left(x,y,\mathbf{a},\frac{\partial a^i}{\partial x},\frac{\partial a^i}{\partial y}\right), \tag{6.2}$$

and such an equation falls into the category of "parabolic" by the above classification. For this reason many three and four dimensional PDEs in physics are referred to as parabolic,

although technically this classification exists only for PDEs in two dimensions. The more general form for such "parabolic-like" differential equations is:

$$\frac{\partial}{\partial x^0} a^i(\mathbf{x}) = f^i\left[\mathbf{x}, \mathbf{a}(\mathbf{x})\right], \tag{6.3}$$

where the functionals $f^i$ may include first or second order partial derivatives of the *transverse* dimensions, $x^{i\neq 0}$. This set of equations can be generalised to allow the functionals $f^i$ to include partial derivatives of any order. Given that the equations are in this general form, they may be propagated in the first order dimension, $x^0$, using one of the algorithms detailed further on.

### 6.1.1 Boundary Conditions

In the case of partial differential equations, the equations must apply either over all of the space spanned by the transverse dimensions, or else over a confined region of this space. Either way, computational resources cannot solve over an infinite region of space, and so boundaries of some sort must be imposed. Therefore, a description of what happens at these boundaries is essential for solving the problem. There are three common types of boundary conditions:

1. **Dirichlet boundary conditions** This is where the value of a component is fixed at a particular boundary:
$$a^i\left(\mathbf{x}|x^k = x^k_\pm\right) = a^i_{k\pm}. \tag{6.4}$$

2. **Neumann boundary conditions** This is where the cross-boundary gradient of a component is fixed at a particular boundary:
$$\frac{\partial}{\partial x^k} a^i\left(\mathbf{x}|x^k = x^k_\pm\right) = b^i_{k\pm}. \tag{6.5}$$

3. **Periodic boundary conditions** This is where the value of a component (and all of its cross-boundary derivatives) is the same at each end of the dimension in question. It is as if this dimension was originally circular, except that it has been cut for the purpose of discretisation.
$$\left(\frac{\partial}{\partial x^k}\right)^n a^i\left(\mathbf{x}|x^k = x^k_+\right) = \left(\frac{\partial}{\partial x^k}\right)^n a^i\left(\mathbf{x}|x^k = x^k_-\right), \quad n = 0, 1, 2... \tag{6.6}$$

If an infinite region of space is absolutely necessary, then it can be mapped to a finite region of space using a transformation such as $x = \tan(\epsilon)$, which maps an infinite range in $x$ to a finite range, $[-\frac{\pi}{2}, +\frac{\pi}{2}]$, in $\epsilon$. There are, of course, a few caveats when performing such a mapping.

Finally, the last boundary condition is the initial state of the components. The values (or form) of all the components must be specified for some point in the dimension of propagation, $x^0$.

## 6.2 Stochastic Equations

Stochastic differential equations also include noise sources, $\xi(\mathbf{x})$, which may be real or complex:

$$\frac{\partial}{\partial x^0}a^i(\mathbf{x}) = f^i\left[\mathbf{x}, \mathbf{a}(\mathbf{x})\right] + g^{ij}\left[\mathbf{a}(\mathbf{x})\right]\xi^j(\mathbf{x}). \tag{6.7}$$

These can make propagation a little more technically challenging, but not impossible, and enables a vast range of complex phenomena to be modelled. As a result an individual propagation path becomes dependent on the underlying noise sources used, and usually many such paths using independent noises are combined and averaged to obtain the desired result. The noise sources are normally delta-correlated in the propagation dimension, though not always in the transverse dimensions, and not always with each other, so that generally:

$$\left\langle \xi^i(\mathbf{x})\xi^j(\mathbf{x}')\right\rangle = \delta(x^{0'} - x^0)C_{ij}(\mathbf{x}'_\perp, \mathbf{x}_\perp); \tag{6.8}$$

$$\left\langle \xi^i(\mathbf{x})\xi^{j*}(\mathbf{x}')\right\rangle = \delta(x^{0'} - x^0)D_{ij}(\mathbf{x}'_\perp, \mathbf{x}_\perp), \tag{6.9}$$

where as usual the subscript $\perp$ denotes the transverse dimensions.

### 6.2.1 Ito vs Stratonovich Calculus

The integration of a stochastic equation as expressed above is not well defined in normal calculus. In fact, there are multiple ways in which such an integration can be defined as the limit of some discrete process, and they are not equivalent. The two main forms of stochastic calculus are Ito calculus and Stratonovich calculus. Ito calculus is better suited to mathematical proofs, but Stratonovich calculus has the major advantage that derivatives follow the normal chain rule. This typically simplifies the numerical technique for solving a Stratonovich equation with a high-order method. Both types of equations arise naturally from modelling physical systems. Fortunately, it is possible to transform a set of equations in Ito calculus to Stratonovich calculus (and vice-versa).

If a set of Ito equations is given by:

$$(I) \qquad \frac{\partial}{\partial x^0}a^i = f^i\left[\mathbf{a}\right] + g^{ij}\left[\mathbf{a}\right]\xi^j, \tag{6.10}$$

then the corresponding set of Stratonovich equations is

$$(S) \qquad \frac{\partial}{\partial x^0}a^i = f^i\left[\mathbf{a}\right] - \frac{1}{2}\sum_{jk} g^{kj}\left[\mathbf{a}\right]\frac{\partial}{\partial a^k}g^{ij}\left[\mathbf{a}\right] + g^{ij}\left[\mathbf{a}(\mathbf{x})\right]\xi^j(\mathbf{x}). \tag{6.11}$$

Numerical integration of stochastic equations is more computationally intensive than deterministic equations of equivalent size and complexity.

## 6.3 Numerical Methods for Differential Equations

We now briefly describe several numerical methods for integrating deterministic and stochastic differential equations.

### 6.3.1 The Euler and Inverse Euler Methods

Consider the ordinary differential equation:

$$\frac{\partial}{\partial t}a(t) = f[t, a], \tag{6.12}$$

where f is some functional of $t$ and/or $a$. Given that a "current" value of $a$ (say $a_n = a(t_n)$) is known then the simplest method to propagate $a$ forward in time is to use the current slope ($f[t_n, a_n]$) to calculate the new value $a_{n+1}$:

$$a_{n+1} = a_n + h\, f[t_n, a_n], \tag{6.13}$$

where $h$ is the time step so that $t_{n+1} = t_n + h$. This is known at the explicit, or *forward time*, Euler method. It is very simple, but not very accurate or stable. Stability can be improved to an extent by thinking in reverse. Rather than using the slope at the current point in time to calculate the next value of $a$, use the slope and the *next* point in time. But this, of course, is not known until the next value of $a$ is known, so the process becomes an iterative, or *implicit* one. Initially the current value of $a$ is used as next value, and then the step:

$$a_{n+1} = a_n + h\, f[t_{n+1}, a_{n+1}], \tag{6.14}$$

is iterated until convergence is obtained. This method is known as the *implicit*, or *backward time*, or *inverse*, Euler method. For certain classes of differential equation, implicit methods are more stable than their explicit equivalents, but overall the implicit Euler method is no more accurate than the explicit Euler method.

The explicit and implicit Euler methods do not account for the second and higher order derivatives of $a$. Thus the error per step is of order $h^2$, which means the error over a given integration interval is of order $h$. Consequently these methods are known as first order methods.

Both of these methods can be used to integrate stochastic equations by treating the noise terms as part of a larger function $f$ which depends on a pseudo-random number which changes each time step. This technique converges to the Ito integral with a convergence of order $h^{1/2}$. In general, the order of the stochastic convergence of a numerical method will be less than or equal to half the order of the deterministic convergence.

### 6.3.2 The Improved Euler Method

Here two evaluations of the derivative $f[t, a]$ are made. The first one is based on the current value $a$, and is used to estimate the next value of $a$ as in the explicit method. This estimated next value is then used to calculate the next value of the derivative, which in turn is used to make a second estimate of the next value of $a$ as in the implicit method. These two estimates are then averaged:

$$a^{(1)}_{n+1} = a_n + h\, f[t_n, a_n]; \tag{6.15}$$

$$a^{(2)}_{n+1} = a_n + h\, f[t_{n+1}, a^{(1)}_{n+1}]; \tag{6.16}$$

$$a_{n+1} = \frac{1}{2}\left(a^{(1)}_{n+1} + a^{(2)}_{n+1}\right). \tag{6.17}$$

The two estimates of $f[t, a]$ account for the second order derivative of $a$, so the error per step is reduced to order $h^3$, and the integral is correct to order $h^2$. This method is also known as the improved Euler-Cauchy method, or Huen's method. It is also known as the second-order Runge-Kutta method.

Unfortunately, this method is not guaranteed to converge for stochastic integration when the noise terms are naively added to the function on the RHS.

### 6.3.3 The Semi-Implicit Method

This method is very similar to the improved Euler method. Essentially the method lies half way between the explicit and implicit methods in that the new value for $a$ is calculated based on the derivative at the mid-point of the step. Similarly to the implicit method, the current value of $a$ is used initially as the mid-point value, $a_m = a(t_m); t_m = t + h/2$, and then the step:

$$a_m = a_n + \frac{1}{2}\, h\, f[t_m, a_m], \tag{6.18}$$

is iterated until convergence is obtained. The next value for $a$ is then:

$$a_{n+1} = 2a_m - a_n. \tag{6.19}$$

Similar to the improved Euler method, the semi-implicit method is also a second order method. It has the added advantage that a naïve inclusion of the noise terms in the function $f$ produces an algorithm that converges to the Stratonovich integral with global error of the order of $h$.

### 6.3.4 The Fourth-Order Runge-Kutta Method

The fourth-order Runge-Kutta method has been the workhorse for numerical modelling for many years. The method uses four evaluations of the derivative to account for the first, second, third, and fourth derivatives—making it a fourth order method. The method is simply:

$$\mathbf{k}_1 = h\, \mathbf{f}[t_n, a_n]; \tag{6.20}$$

$$\mathbf{k}_2 = h\, \mathbf{f}[\frac{1}{2}(t_n + t_{n+1}), \mathbf{a}_n + \frac{1}{2}\mathbf{k}_1]; \tag{6.21}$$

$$\mathbf{k}_3 = h\, \mathbf{f}[\frac{1}{2}(t_n + t_{n+1}), \mathbf{a}_n + \frac{1}{2}\mathbf{k}_2]; \tag{6.22}$$

$$\mathbf{k}_4 = h\, \mathbf{f}[t_{n+1}, \mathbf{a}_n + \mathbf{k}_3]; \tag{6.23}$$

$$\mathbf{a}_{n+1} = \mathbf{a}_n + \frac{1}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right). \tag{6.24}$$

$$\tag{6.25}$$

Higher order methods are possible, but usually the extra accuracy does not warrant the extra computation. In other words, this method usually achieves the fastest evaluation for a given propagation accuracy. Note that even an optimised implementation of this algorithm will require about three times as much memory as a lower order method.

Although empirically there is a subclass of Stratonovich stochastic problems that converge with this algorithm to second order, there is no guarantee of convergence for stochastic equations.

### 6.3.5 The Adaptive Fourth-Order Runge-Kutta Method

For problems that exhibit very diverse behavior of the solution, say with slow change for large parts of the propagation and rapid variations in others, it is very useful to employ an algorithm that can adjust its stepsize. This allows the simulation to speed through smooth uninteresting countryside in a few great strides but to tiptoe with many small steps through treacherous terrain.

The adaptive stepsize method implemented in **xmds** in an embedded fourth-fifth order Runge-Kutta method (ARK45). It takes advantage of the fact that it is possible to combine certain six function evaluations that can result in a fifth order Runge-Kutta method in another way to give a result that is accurate to fourth order. The ARK45 algorithm calculates the forth and fifth order solution and uses the difference between them as an estimate of the current discretisation error.

The structure is:

$$\begin{aligned}
\mathbf{k}_1 &= h\,\mathbf{f}[t_n, a_n];\\
\mathbf{k}_2 &= h\,\mathbf{f}[t_n + a_2 h, \mathbf{a}_n + b_{21}\mathbf{k}_1];\\
\mathbf{k}_3 &= h\,\mathbf{f}[t_n + a_3 h, \mathbf{a}_n + b_{31}\mathbf{k}_1 + b_{32}\mathbf{k}_2];\\
\mathbf{k}_4 &= h\,\mathbf{f}[t_n + a_4 h, \mathbf{a}_n + b_{41}\mathbf{k}_1 + b_{42}\mathbf{k}_2 + b_{43}\mathbf{k}_3];\\
\mathbf{k}_5 &= h\,\mathbf{f}[t_n + a_5 h, \mathbf{a}_n + b_{51}\mathbf{k}_1 + b_{52}\mathbf{k}_2 + b_{53}\mathbf{k}_3 + b_{54}\mathbf{k}_4];\\
\mathbf{k}_6 &= h\,\mathbf{f}[t_n + a_6 h, \mathbf{a}_n + b_{61}\mathbf{k}_1 + b_{62}\mathbf{k}_2 + b_{63}\mathbf{k}_3 + b_{64}\mathbf{k}_4 + b_{65}\mathbf{k}_5];\\
\mathbf{a}_{n+1} &= \mathbf{a}_n + c_1\mathbf{k}_1 + c_2\mathbf{k}_2 + c_3\mathbf{k}_3 + c_4\mathbf{k}_4 + c_5\mathbf{k}_5 + c_6\mathbf{k}_6 + \mathcal{O}(h^6);\\
\mathbf{a}^*_{n+1} &= \mathbf{a}_n + c^*_1\mathbf{k}_1 + c^*_2\mathbf{k}_2 + c^*_3\mathbf{k}_3 + c^*_4\mathbf{k}_4 + c^*_5\mathbf{k}_5 + c^*_6\mathbf{k}_6 + \mathcal{O}(h^5).
\end{aligned} \tag{6.26}$$

Here $\mathbf{a}_{n+1}$ is the fifth order- and $\mathbf{a}^*_{n+1}$ the fourth order solution. The coefficients that appear are listed in table 6.1. To adjust the time step **xmds** calculates the relative difference between the two solutions $\Delta_m$. If partial differential equations are being solved it will determine the maximum of the relative errors of all grid points, omitting points where the function is less than $c_{thresh} \times M[a]$, where $c_{thresh}$ is a small threshold value (see chapter 11) and $M[a]$ is the peak value of the function across the grid. If multiple functions are present, each peak value is determined separately.

If the prescribed accuracy is $\Delta_{\text{tol}}$, a time step of size $\delta\tau_n$ is accepted if $\Delta_m \leq \Delta_{\text{tol}}$. It is rejected if $\Delta_m > \Delta_{\text{tol}}$, and the step must be calculated again. The size of the next time step $\delta\tau_{n+1}$ is determined by:

$$\delta\tau_{n+1} = \begin{cases} 0.92\delta\tau_n \left|\dfrac{\Delta_{\text{tol}}}{\Delta_m}\right|^{1/5} & \text{if } \Delta_m \leq \Delta_{\text{tol}}, \\ \\ 0.92\delta\tau_n \left|\dfrac{\Delta_{\text{tol}}}{\Delta_m}\right|^{1/4} & \text{if } \Delta_m > \Delta_{\text{tol}}. \end{cases} \tag{6.27}$$

| $i$ | $a_i$ | $b_{ij}$ | | | | | $c_i$ | $c_i^*$ |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | | | | $\frac{37}{378}$ | $\frac{2825}{27648}$ |
| 2 | $\frac{1}{5}$ | $\frac{1}{5}$ | | | | | 0 | 0 |
| 3 | $\frac{3}{10}$ | $\frac{3}{40}$ | $\frac{9}{40}$ | | | | $\frac{250}{621}$ | $\frac{18575}{48384}$ |
| 4 | $\frac{3}{5}$ | $\frac{3}{10}$ | $-\frac{9}{10}$ | $\frac{6}{5}$ | | | $\frac{125}{594}$ | $\frac{13525}{55296}$ |
| 5 | 1 | $-\frac{11}{54}$ | $\frac{5}{2}$ | $-\frac{70}{27}$ | $\frac{35}{27}$ | | 0 | $\frac{277}{14336}$ |
| 6 | $\frac{7}{8}$ | $\frac{1631}{55296}$ | $\frac{175}{512}$ | $\frac{575}{13824}$ | $\frac{44275}{110592}$ | $\frac{253}{4096}$ | $\frac{512}{1771}$ | $\frac{1}{4}$ |
| j = | | 1 | 2 | 3 | 4 | 5 | | |

Table 6.1: Cash-Karp parameters for the embedded Runga-Kutta method.

This algorithm will consume even more memory than the fourth-order Runge-Kutta, but this easily pays off in situations where the rate of change of the solution varies a lot, or when the appropriate step size is unknown.

As with the fourth-order Runge-Kutta there is no guarantee of convergence for stochastic equations.

## 6.4 Numerical Methods for Partial Differential Equations

The methods in the previous section all deal with ordinary differential equations quite well, but partial differential equations are considerably more complex, since the derivative depends also upon the values for $a$ at other points in the transverse space. In principle, once the transverse spaces have been divided into a finite lattice, sets of partial differential equations are just larger sets of ordinary differential equations, but there are many subtleties involved with achieving this discretisation.

The critical issue is how the evolution due to the transverse derivatives is performed. This issue divides algorithms into two classes: explicit picture and interaction picture. We begin here by explaining just how the transverse derivatives may be calculated, and then outline how the Semi-Implicit and the Runge-Kutta algorithms can be applied in each of these cases.

### 6.4.1 Evaluating Transverse Derivatives

The transverse dimensions must be discretised to form a lattice. Lattices with more points enable more detail, but also take up more memory, and more computational resource. The transverse derivatives must be evaluated at each lattice point, and there must be some method for doing so that uses the values of the field from the other lattice points. There are two main methods for doing this.

Firstly, there is the **matrix** method. If we let $a_k$ represent the discretised values of $a(x)$ at the lattice points $x_k$, then we may evaluate the first order derivative like so:

$$\left(\frac{\partial a(x)}{\partial x}\right)_{x_k} = \frac{a_{k+1} - a_{k-1}}{x_{k+1} - x_{k-1}} \tag{6.28}$$

$$= M_{jk} a_j, \quad k \neq 1, n. \tag{6.29}$$

Note the symmetry of the weightings about the point where the derivative is being evaluated, causing the method to be *space centered*. Note also that, unless periodic boundary conditions are specified, it is not not possible to implement a space centered method at the ends of the lattice. The best that can be done is to implement a non-space centered method at the ends, and insist that the derivatives vanish at the lattice boundaries for the technique to be valid. $\mathbf{M}$ may be expressed as a tri-diagonal square matrix, which for periodic boundary conditions is:

$$\mathbf{M} = \frac{1}{2\Delta x} \begin{bmatrix} 0 & 1 & 0 & 0 & ... & ... & -1 \\ -1 & 0 & 1 & 0 & ... & ... & 0 \\ 0 & -1 & 0 & 1 & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & -1 & 0 & 1 & 0 \\ 0 & ... & ... & 0 & -1 & 0 & 1 \\ 1 & ... & ... & 0 & 0 & -1 & 0 \end{bmatrix}. \tag{6.30}$$

And similarly for evaluating the second derivative we might use:

$$\mathbf{M} = \frac{1}{(\Delta x)^2} \begin{bmatrix} -2 & 1 & 0 & 0 & ... & ... & 1 \\ 1 & -2 & 1 & 0 & ... & ... & 0 \\ 0 & 1 & -2 & 1 & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & ... & ... & 1 & -2 & 1 & 0 \\ 0 & ... & ... & 0 & 1 & -2 & 1 \\ 1 & ... & ... & 0 & 0 & 1 & -2 \end{bmatrix}. \tag{6.31}$$

These methods evaluate the derivatives using only the adjacent points in space, but in principle it is possible to use progressively more distant neighbours to produce an estimate of each derivative which takes into account contributions from higher order derivatives. In practice, these methods are used rarely due to the availability of a method based on the **Fourier Transform**. The partial derivate with respect to dimension $x^j$ is calculated quite simply using Equation (6.32):

$$\frac{\partial}{\partial x^j} a(\mathbf{x}) = \frac{\partial}{\partial x^j} \frac{1}{(2\pi)^{\frac{N}{2}}} \int d\mathbf{k} e^{i\mathbf{k}\cdot\mathbf{x}} \tilde{a}(\mathbf{k})$$

$$= \frac{1}{(2\pi)^{\frac{N}{2}}} \int d\mathbf{k} e^{i\mathbf{k}\cdot\mathbf{x}} (ik^j) \tilde{a}(\mathbf{k}). \tag{6.32}$$

This method can equally be applied to discrete data. Firstly a discrete Fourier Transform is applied to the field data $a_k$, then the data is multiplied point by point by the corresponding

$ik^j$ values, and then the discrete inverse Fourier Transform is applied to obtain the $a'_k$ data. Extending this method for higher order derivatives is trivial. Equation (6.33) illustrates some example mappings:

$$\begin{aligned} f(k_x, k_y, ...) &\mapsto f(k_x, k_y, ...); \\ \frac{\partial}{\partial x} &\mapsto ik_x; \\ \frac{\partial^2}{\partial x^2} &\mapsto (ik_x)^2 = -k_x^2; \\ \frac{\partial^2}{\partial x \partial y} &\mapsto (ik_x)(ik_y) = -k_x k_y. \end{aligned} \tag{6.33}$$

This method becomes computationally more efficient than the matrix method when a number of higher order derivatives are desired, for example when a Laplacian operator is being applied in multiple transverse dimensions. The main limitations of this method are that it implies periodic boundary conditions and that it cannot be implemented on a non-regular grid.

### 6.4.2 Explicit Picture Methods

In the explicit picture, all linear and nonlinear operators are explicitly calculated and summed to generate the total derivative for each field component, whereas in the interaction picture linear operators are dealt with separately to nonlinear operators.

Once a set of partial differential equations has been discretised to a finite lattice in the transverse dimensions, it is formally equivalent to a large set of coupled ordinary differential equations, and therefore it can be solved by higher order methods such as the semi-implicit method of Section 6.3.3, or the fourth order Runge-Kutta (RK4) method of Section 6.3.4. Partial derivatives of the fields can be determined by Crank-Nicholson methods or spectral methods. Thus explicit picture methods are capable of solving the fully fledged generalised PDE:

$$\frac{\partial}{\partial x^0}\mathbf{a}(\mathbf{x}) = \mathcal{N}\left(\mathbf{x},\ \mathbf{a}(\mathbf{x}), \mathbf{p}(\mathbf{x}),\ \xi(\mathbf{x})\right), \tag{6.34}$$

$$p^i(\mathbf{x}) = \mathcal{F}^{-1}\left[\Sigma_j \mathcal{L}^{ij}\left(x^0, \mathbf{k}_\perp\right)\mathcal{F}\left[a^j(\mathbf{x})\right]\right]. \tag{6.35}$$

This is the PDE that is listed in Equation (8.3) with the cross propagating vector **b** omitted. Refer to Chapter 8 for a full explanation of the variables. The important thing to note is that the total derivative of the vector **a** is expressed in terms of the general nonlinear functionals $\mathcal{N}$, and that this form allows for nonlinear partial derivatives that may have spatial dependence, for example $\frac{1}{r}\left(\frac{\partial}{\partial r}\right)^2$.

### 6.4.3 The Semi-Implicit method in the Explicit Picture

Here we use the Fourier Transform (or Spectral) method to determine the transverse derivatives. The procedure, after Section 6.3.3, is:

1. Calculate $\xi$ if required.
2. $x^0 = x^0 + \frac{1}{2}h$
3. $\mathbf{a}_I = \mathbf{a}$
4. For N-1 iterations do:
   (a) $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[a\right]\right]$
   (b) $\mathbf{a} = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}, \mathbf{p}, \xi\right)$
   (c) $\mathbf{a} = \mathbf{a}_I + \frac{1}{2}\mathbf{a}$
5. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[a\right]\right]$
6. $\mathbf{a} = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}, \mathbf{p}, \xi\right)$
7. $\mathbf{a} = \mathbf{a}_I + \mathbf{a}$
8. $x^0 = x^0 + \frac{1}{2}h$

Here an extra copy of the field is needed to store the initial $\mathbf{a}$ into $\mathbf{a}_I$, and a similar size vector is needed for the derivatives $\mathbf{p}$.

### 6.4.4 The Fourth Order Runge-Kutta Method in the Explicit Picture

Again we use the Fourier Transform (or Spectral) method to determine the transverse derivatives. The procedure, after Section 6.3.4, then is:

1. Calculate $\xi$ if required.
2. $\mathbf{a}_K = \mathbf{a}$
3. $\mathbf{a}_I = \mathbf{a}$
4. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
5. $\mathbf{a}_K = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_K, \mathbf{p}, \xi\right)$
6. $\mathbf{a} = \mathbf{a} + \frac{1}{6}\mathbf{a}_K$
7. $x^0 = x^0 + \frac{1}{2}h$
8. $\mathbf{a}_K = \mathbf{a}_I + \frac{1}{2}\mathbf{a}_K$
9. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
10. $\mathbf{a}_K = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_K, \mathbf{p}, \xi\right)$
11. $\mathbf{a} = \mathbf{a} + \frac{1}{3}\mathbf{a}_K$

12. $\mathbf{a}_K = \mathbf{a}_I + \frac{1}{2}\mathbf{a}_K$
13. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
14. $\mathbf{a}_K = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_K, \mathbf{p}, \xi\right)$
15. $\mathbf{a} = \mathbf{a} + \frac{1}{3}\mathbf{a}_K$
16. $x^0 = x^0 + \frac{1}{2}h$
17. $\mathbf{a} = \mathbf{a}_I + \mathbf{a}$
18. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
19. $\mathbf{a}_K = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_K, \mathbf{p}, \xi\right)$
20. $\mathbf{a} = \mathbf{a} + \frac{1}{6}\mathbf{a}_K$

This method requires two extra copies of the field. One, $\mathbf{a}_I$, for the initial field value, and the other, $\mathbf{a}_K$, to act as a working field while the original vector $\mathbf{a}$ collects the derivative contributions. This saves the final step of copying $\mathbf{a}_K$ back into $\mathbf{a}$ that would had to have been performed if $\mathbf{a}_K$ had collected the derivative contributions. It also requires a similar size vector for the derivatives $\mathbf{p}$. On most systems where the calculation is not memory-limited, this method tends to achieve a desired accuracy many times faster than the semi-implicit method.

### 6.4.5 The Fourth/ Fifth Order adaptive Runge-Kutta Method in the Explicit Picture

Once more we use the Fourier Transform (or Spectral) method to determine the transverse derivatives. In order to minimize memory usage the current contents of the arrays containing the final solution are reused to calculate some of the k-vectors. The optimized procedure, after Section 6.3.5, then is:

1. Calculate $\xi$ if required.
2. $\mathbf{a}_K = \mathbf{a}$
3. $\mathbf{a}_I = \mathbf{a}$
4. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
5. $\mathbf{a}_K = h\,\mathcal{N}\left(x^0, \mathbf{a}_K, \mathbf{p}, \xi\right)$
6. $\mathbf{a} = \mathbf{a} + c_1\mathbf{a}_K$
7. $\mathbf{a}^* = \mathbf{a}^* + c_1^*\mathbf{a}_K$
8. $x^0 = x^0 + a_2 h$

9. $\mathbf{a}_K = \mathbf{a}_I + b_{21}\mathbf{a}_K$
10. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
11. $\mathbf{a}_K = h\,\mathcal{N}\left(x^0, \mathbf{a}_K, \mathbf{p}, \xi\right)$
12. $\mathbf{a}_J = (1 - b_{31}/c_1)\mathbf{a}_I + b_{31}/c_1\mathbf{a} + b_{32}\mathbf{a}_K$
13. $x^0 = x^0 + (a_3 - a_2)h$
    (Note: $c_2 = c_2^* = 0$)
14. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
15. $\mathbf{a}_J = h\,\mathcal{N}\left(x^0, \mathbf{a}_J, \mathbf{p}, \xi\right)$
16. $\mathbf{a}_L = (1 - b_{41}/c_1)\mathbf{a}_I + b_{41}/c_1\mathbf{a} + b_{42}\mathbf{a}_K + b_{43}\mathbf{a}_J$
17. $\mathbf{a} = \mathbf{a} + c_3\mathbf{a}_J$
18. $\mathbf{a}^* = \mathbf{a}^* + c_3^*\mathbf{a}_J$
19. $x^0 = x^0 + (a_4 - a_3)h$

20. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
21. $\mathbf{a}_L = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_L, \mathbf{p}, \xi\right)$
22. $\mathbf{a} = \mathbf{a} + c_4\mathbf{a}_L$
23. $\mathbf{a}^* = \mathbf{a}^* + c_4^*\mathbf{a}_L$
24. $\mathbf{a}_L = (1 - b_{51}/c_1)\mathbf{a}_I + b_{51}/c_1\mathbf{a} + b_{52}\mathbf{a}_K + (b_{53} - b_{51}c_3/c_1)\mathbf{a}_J + (b_{54} - b_{51}c_4/c_1)\mathbf{a}_L$
25. $x^0 = x^0 + (a_5 - a_4)h$

26. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
27. $\mathbf{a}_L = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_L, \mathbf{p}, \xi\right)$
28. $\mathbf{a}^* = \mathbf{a}^* + c_5^*\mathbf{a}_L$
    (Note: $c_5 = 0$)
29. $\mathbf{a}_L = f_1\mathbf{a}_I + f_2\mathbf{a}_K + f_3\mathbf{a}_J + f_4\mathbf{a} + f_5\mathbf{a}_L + f_6\mathbf{a}^*$
30. $x^0 = x^0 + (a_6 - a_5)h$

31. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_K\right]\right]$
32. $\mathbf{a}_L = h\,\mathcal{N}\left(\mathbf{x}, \mathbf{a}_L, \mathbf{p}, \xi\right)$

33. $\mathbf{a} = \mathbf{a} + c_6 \mathbf{a}_J$
34. $\mathbf{a}^* = \mathbf{a}^* + c_6^* \mathbf{a}_J$

The $f_i$ coefficients for the last function evaluation are derived from the original Cash-Karp coefficients (see table 6.1) in the following way:

$$\begin{aligned} g &= c_1 c_4^* - c_1^* c_4 \\ f_1 &= 1 + (b_{64}(c_1^* - c_1) + b_{61}(c_4 - c_4^*))/g \\ f_2 &= b_{62} \\ f_3 &= b_{63} + (b_{64}(c_1^* - c_3) + b_{61}(c_3^* c_4 - c_3 c_4^*))/g \\ f_4 &= (b_{61} c_4^* - b_{64} c_1^*)/g \\ f_5 &= b_{65} + c_5^*(b_{61} c_4 - b_{64} c_1)/g \\ f_6 &= (b_{64} c_1 - b_{61} c_4)/g \end{aligned} \tag{6.36}$$

This method requires six extra copies of the field. One, $\mathbf{a}_I$, for the initial field value, three $\mathbf{a}_K$, $\mathbf{a}_J$, $\mathbf{a}_L$, to act as a working fields while the original vector $\mathbf{a}$ and $\mathbf{a}^*$ collect the derivative contributions. It also requires a similar size vector for the derivatives $\mathbf{p}$.

### 6.4.6 The Ninth Order Runge-Kutta Method in the Explicit Picture

We continue to use the Fourier Transform (or Spectral) method to determine the transverse derivatives. The procedure is

1. Calculate $\xi$ if required.
2. $x^0 = x^0 + a_1 h$
3. $\mathbf{a}_a = \mathbf{a}$
4. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_a\right]\right]$
5. $\mathbf{a}_a = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_a, \mathbf{p}, \xi\right)$
6. $x^0 = x^0 + (a_2 - a_1)h$
7. $\mathbf{a}_b = \mathbf{a} + b_{2,1}\mathbf{a}_a$
8. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_b\right]\right]$
9. $\mathbf{a}_b = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_b, \mathbf{p}, \xi\right)$
10. $x^0 = x^0 + (a_3 - a_2)h$
11. $\mathbf{a}_c = \mathbf{a} + b_{3,1}\mathbf{a}_a + b_{3,2}\mathbf{a}_b$

12. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_c\right]\right]$
13. $\mathbf{a}_c = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_c, \mathbf{p}, \xi\right)$
14. $x^0 = x^0 + (a_4 - a_3)h$
15. $\mathbf{a}_d = \mathbf{a} + b_{4,1}\mathbf{a}_a + b_{4,2}\mathbf{a}_b + b_{4,3}\mathbf{a}_c$
16. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_d\right]\right]$
17. $\mathbf{a}_d = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_d, \mathbf{p}, \xi\right)$
18. $x^0 = x^0 + (a_5 - a_4)h$
19. $\mathbf{a}_e = \mathbf{a} + b_{5,1}\mathbf{a}_a + b_{5,2}\mathbf{a}_b + b_{5,3}\mathbf{a}_c + b_{5,4}\mathbf{a}_d$
20. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_e\right]\right]$
21. $\mathbf{a}_e = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_e, \mathbf{p}, \xi\right)$
22. $x^0 = x^0 + (a_6 - a_5)h$
23. $\mathbf{a}_i = \mathbf{a} + b_{6,1}\mathbf{a}_a + b_{6,2}\mathbf{a}_b + b_{6,3}\mathbf{a}_c + b_{6,4}\mathbf{a}_d + b_{6,5}\mathbf{a}_e$
24. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_f\right]\right]$
25. $\mathbf{a}_i = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_i, \mathbf{p}, \xi\right)$
26. $x^0 = x^0 + (a_7 - a_6)h$
27. $\mathbf{a}_j = \mathbf{a} + b_{7,1}\mathbf{a}_a + b_{7,2}\mathbf{a}_b + b_{7,3}\mathbf{a}_c + b_{7,4}\mathbf{a}_d + b_{7,5}\mathbf{a}_e + b_{7,6}\mathbf{a}_i$
28. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_j\right]\right]$
29. $\mathbf{a}_j = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_j, \mathbf{p}, \xi\right)$
30. $x^0 = x^0 + (a_8 - a_7)h$
31. $\mathbf{a}_b = \mathbf{a} + b_{8,1}\mathbf{a}_a + b_{8,6}\mathbf{a}_i + b_{8,7}\mathbf{a}_j$
32. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_b\right]\right]$
33. $\mathbf{a}_b = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_b, \mathbf{p}, \xi\right)$
34. $x^0 = x^0 + (a_9 - a_8)h$
35. $\mathbf{a}_c = \mathbf{a} + b_{9,1}\mathbf{a}_a + b_{9,6}\mathbf{a}_i + b_{9,7}\mathbf{a}_j + b_{9,8}\mathbf{a}_b$
36. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_c\right]\right]$
37. $\mathbf{a}_c = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_c, \mathbf{p}, \xi\right)$
38. $x^0 = x^0 + (a_{10} - a_9))h$

39. $\mathbf{a}_d = \mathbf{a} + b_{10,1}\mathbf{a}_a + b_{10,6}\mathbf{a}_i + b_{10,7}\mathbf{a}_j + b_{10,8}\mathbf{a}_b + b_{10,9}\mathbf{a}_c$
40. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_d\right]\right]$
41. $\mathbf{a}_d = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_d, \mathbf{p}, \xi\right)$
42. $x^0 = x^0 + (a_{11} - a_{10})h$
43. $\mathbf{a}_e = \mathbf{a} + b_{11,1}\mathbf{a}_a + b_{11,6}\mathbf{a}_i + b_{11,7}\mathbf{a}_j + b_{11,8}\mathbf{a}_b + b_{11,9}\mathbf{a}_c + b_{11,10}\mathbf{a}_d$
44. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_e\right]\right]$
45. $\mathbf{a}_e = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_e, \mathbf{p}, \xi\right)$
46. $x^0 = x^0 + (a_{12} - a_{11})h$
47. $\mathbf{a}_f = \mathbf{a} + b_{12,1}\mathbf{a}_a + b_{12,6}\mathbf{a}_i + b_{12,7}\mathbf{a}_j + b_{12,8}\mathbf{a}_b + b_{12,9}\mathbf{a}_c + b_{12,10}\mathbf{a}_d + b_{12,11}\mathbf{a}_e$
48. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_f\right]\right]$
49. $\mathbf{a}_f = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_f, \mathbf{p}, \xi\right)$
50. $x^0 = x^0 + (a_{13} - a_{12})h$
51. $\mathbf{a}_g = \mathbf{a} + b_{13,1}\mathbf{a}_a + b_{13,6}\mathbf{a}_i + b_{13,7}\mathbf{a}_j + b_{13,8}\mathbf{a}_b + b_{13,9}\mathbf{a}_c + b_{13,10}\mathbf{a}_d + b_{13,11}\mathbf{a}_e + b_{13,12}\mathbf{a}_f$
52. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_g\right]\right]$
53. $\mathbf{a}_g = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_g, \mathbf{p}, \xi\right)$
54. $x^0 = x^0 + (a_{14} - a_{13})h$
55. $\mathbf{a}_h = \mathbf{a} + b_{14,1}\mathbf{a}_a + b_{14,6}\mathbf{a}_i + b_{14,7}\mathbf{a}_j + b_{14,8}\mathbf{a}_b + b_{14,9}\mathbf{a}_c + b_{14,10}\mathbf{a}_d + b_{14,11}\mathbf{a}_e + b_{14,12}\mathbf{a}_f + b_{14,13}\mathbf{a}_g$
56. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_h\right]\right]$
57. $\mathbf{a}_h = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_h, \mathbf{p}, \xi\right)$
58. $\mathbf{a}_i = \mathbf{a} + b_{15,1}\mathbf{a}_a + b_{15,6}\mathbf{a}_i + b_{15,7}\mathbf{a}_j + b_{15,8}\mathbf{a}_b + b_{15,9}\mathbf{a}_c + b_{15,10}\mathbf{a}_d + b_{15,11}\mathbf{a}_e + b_{15,12}\mathbf{a}_f + b_{15,13}\mathbf{a}_g + b_{15,14}\mathbf{a}_h$
59. $\mathbf{a}_j = (1 - b_{16,6}/b_{15,6})\mathbf{a} + (b_{16,1} - b_{15,1}b_{16,6}/b_{15,6})\mathbf{a}_a + (b_{16,6}/b_{15,6})\mathbf{a}_i + (b_{16,7} - b_{15,7}b_{16,6}/b_{15,6})\mathbf{a}_j + (b_{16,8} - b_{16,8}b_{16,6}/b_{15,6})\mathbf{a}_b + (b_{16,9} - b_{15,9}b_{16,6}/b_{15,6})\mathbf{a}_c + (b_{16,10} - b_{15,10}b_{16,6}/b_{15,6})\mathbf{a}_d + (b_{16,11} - b_{15,11}b_{16,6}/b_{15,6})\mathbf{a}_e + (b_{16,12} - b_{15,12}b_{16,6}/b_{15,6})\mathbf{a}_f + (b_{16,13} - b_{15,13}b_{16,6}/b_{15,6})\mathbf{a}_g + (b_{16,14} - b_{15,14}b_{16,6}/b_{15,6})\mathbf{a}_h$
60. $x^0 = x^0 + (a_{15} - a_{14})h$
61. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_i\right]\right]$
62. $\mathbf{a}_i = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_i, \mathbf{p}, \xi\right)$

63. $x^0 = x^0 + (a_{16} - a_{15})h$
64. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_j\right]\right]$
65. $\mathbf{a}_j = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_j, \mathbf{p}, \xi\right)$
66. $\mathbf{a} = \mathbf{a} + c_1\mathbf{a}_a + c_8\mathbf{a}_b + c_9\mathbf{a}_c + c_{10}\mathbf{a}_d + c_{11}\mathbf{a}_e + c_{12}\mathbf{a}_f + c_{13}\mathbf{a}_g + c_{14}\mathbf{a}_h + c_{15}\mathbf{a}_i + c_{16}\mathbf{a}_j$

This method requires 10 extra copies of the field, $\mathbf{a}_a$ to $\mathbf{j}_j$. Many of the $a_i$ $b_{ij}$ and $c_i$ constants are zero in the Eight/Ninth order Runge Kutta method are zero, meaning not all fields produced at each midstep are required all the time, this allows us to reduce the total number of fields required significantly. The 15th and 16th steps were merged together as well to reduce the total fields needed in memory. Stochastic problems when solved with this methods will achieve a significant performance boost many orders or magnitude greater than extra memory use.

### 6.4.7 The Eighth/Ninth Order adaptive Runge-Kutta Method in the Explicit Picture

We continue to use the Fourier Transform (or Spectral) method to determine the transverse derivatives. The procedure is

1. Calculate $\xi$ if required.
2. $x^0 = x^0 + a_1 h$
3. $\mathbf{a}_a = \mathbf{a}$
4. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_a\right]\right]$
5. $\mathbf{a}_a = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_a, \mathbf{p}, \xi\right)$
6. $x^0 = x^0 + (a_2 - a_1)h$
7. $\mathbf{a}_b = \mathbf{a} + b_{2,1}\mathbf{a}_a$
8. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_b\right]\right]$
9. $\mathbf{a}_b = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_b, \mathbf{p}, \xi\right)$
10. $x^0 = x^0 + (a_3 - a_2)h$
11. $\mathbf{a}_c = \mathbf{a} + b_{3,1}\mathbf{a}_a + b_{3,2}\mathbf{a}_b$
12. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_c\right]\right]$
13. $\mathbf{a}_c = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_c, \mathbf{p}, \xi\right)$
14. $x^0 = x^0 + (a_4 - a_3)h$

15. $\mathbf{a}_d = \mathbf{a} + b_{4,1}\mathbf{a}_a + b_{4,2}\mathbf{a}_b + b_{4,3}\mathbf{a}_c$
16. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_d\right]\right]$
17. $\mathbf{a}_d = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_d, \mathbf{p}, \xi\right)$
18. $x^0 = x^0 + (a_5 - a_4)h$
19. $\mathbf{a}_e = \mathbf{a} + b_{5,1}\mathbf{a}_a + b_{5,2}\mathbf{a}_b + b_{5,3}\mathbf{a}_c + b_{5,4}\mathbf{a}_d$
20. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_e\right]\right]$
21. $\mathbf{a}_e = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_e, \mathbf{p}, \xi\right)$
22. $x^0 = x^0 + (a_6 - a_5)h$
23. $\mathbf{a}_i = \mathbf{a} + b_{6,1}\mathbf{a}_a + b_{6,2}\mathbf{a}_b + b_{6,3}\mathbf{a}_c + b_{6,4}\mathbf{a}_d + b_{6,5}\mathbf{a}_e$
24. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_f\right]\right]$
25. $\mathbf{a}_i = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_i, \mathbf{p}, \xi\right)$
26. $x^0 = x^0 + (a_7 - a_6)h$
27. $\mathbf{a}_j = \mathbf{a} + b_{7,1}\mathbf{a}_a + b_{7,2}\mathbf{a}_b + b_{7,3}\mathbf{a}_c + b_{7,4}\mathbf{a}_d + b_{7,5}\mathbf{a}_e + b_{7,6}\mathbf{a}_i$
28. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_j\right]\right]$
29. $\mathbf{a}_j = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_j, \mathbf{p}, \xi\right)$
30. $x^0 = x^0 + (a_8 - a_7)h$
31. $\mathbf{a}_b = \mathbf{a} + b_{8,1}\mathbf{a}_a + b_{8,6}\mathbf{a}_i + b_{8,7}\mathbf{a}_j$
32. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_b\right]\right]$
33. $\mathbf{a}_b = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_b, \mathbf{p}, \xi\right)$
34. $x^0 = x^0 + (a_9 - a_8)h$
35. $\mathbf{a}_c = \mathbf{a} + b_{9,1}\mathbf{a}_a + b_{9,6}\mathbf{a}_i + b_{9,7}\mathbf{a}_j + b_{9,8}\mathbf{a}_b$
36. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_c\right]\right]$
37. $\mathbf{a}_c = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_c, \mathbf{p}, \xi\right)$
38. $x^0 = x^0 + (a_{10} - a_9))h$
39. $\mathbf{a}_d = \mathbf{a} + b_{10,1}\mathbf{a}_a + b_{10,6}\mathbf{a}_i + b_{10,7}\mathbf{a}_j + b_{10,8}\mathbf{a}_b + b_{10,9}\mathbf{a}_c$
40. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_d\right]\right]$
41. $\mathbf{a}_d = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_d, \mathbf{p}, \xi\right)$

42. $x^0 = x^0 + (a_{11} - a_{10})h$
43. $\mathbf{a}_e = \mathbf{a} + b_{11,1}\mathbf{a}_a + b_{11,6}\mathbf{a}_i + b_{11,7}\mathbf{a}_j + b_{11,8}\mathbf{a}_b + b_{11,9}\mathbf{a}_c + b_{11,10}\mathbf{a}_d$
44. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_e\right]\right]$
45. $\mathbf{a}_e = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_e, \mathbf{p}, \xi\right)$
46. $x^0 = x^0 + (a_{12} - a_{11})h$
47. $\mathbf{a}_f = \mathbf{a} + b_{12,1}\mathbf{a}_a + b_{12,6}\mathbf{a}_i + b_{12,7}\mathbf{a}_j + b_{12,8}\mathbf{a}_b + b_{12,9}\mathbf{a}_c + b_{12,10}\mathbf{a}_d + b_{12,11}\mathbf{a}_e$
48. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_f\right]\right]$
49. $\mathbf{a}_f = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_f, \mathbf{p}, \xi\right)$
50. $x^0 = x^0 + (a_{13} - a_{12})h$
51. $\mathbf{a}_g = \mathbf{a} + b_{13,1}\mathbf{a}_a + b_{13,6}\mathbf{a}_i + b_{13,7}\mathbf{a}_j + b_{13,8}\mathbf{a}_b + b_{13,9}\mathbf{a}_c + b_{13,10}\mathbf{a}_d + b_{13,11}\mathbf{a}_e + b_{13,12}\mathbf{a}_f$
52. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_g\right]\right]$
53. $\mathbf{a}_g = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_g, \mathbf{p}, \xi\right)$
54. $x^0 = x^0 + (a_{14} - a_{13})h$
55. $\mathbf{a}_h = \mathbf{a} + b_{14,1}\mathbf{a}_a + b_{14,6}\mathbf{a}_i + b_{14,7}\mathbf{a}_j + b_{14,8}\mathbf{a}_b + b_{14,9}\mathbf{a}_c + b_{14,10}\mathbf{a}_d + b_{14,11}\mathbf{a}_e + b_{14,12}\mathbf{a}_f + b_{14,13}\mathbf{a}_g$
56. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_h\right]\right]$
57. $\mathbf{a}_h = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_h, \mathbf{p}, \xi\right)$
58. $\mathbf{a}_i = \mathbf{a} + b_{15,1}\mathbf{a}_a + b_{15,6}\mathbf{a}_i + b_{15,7}\mathbf{a}_j + b_{15,8}\mathbf{a}_b + b_{15,9}\mathbf{a}_c + b_{15,10}\mathbf{a}_d + b_{15,11}\mathbf{a}_e + b_{15,12}\mathbf{a}_f + b_{15,13}\mathbf{a}_g + b_{15,14}\mathbf{a}_h$
59. $\mathbf{a}_j = (1 - b_{16,6}/b_{15,6})\mathbf{a} + (b_{16,1} - b_{15,1}b_{16,6}/b_{15,6})\mathbf{a}_a + (b_{16,6}/b_{15,6})\mathbf{a}_i + (b_{16,7} - b_{15,7}b_{16,6}/b_{15,6})\mathbf{a}_j + (b_{16,8} - b_{16,8}b_{16,6}/b_{15,6})\mathbf{a}_b + (b_{16,9} - b_{15,9}b_{16,6}/b_{15,6})\mathbf{a}_c + (b_{16,10} - b_{15,10}b_{16,6}/b_{15,6})\mathbf{a}_d + (b_{16,11} - b_{15,11}b_{16,6}/b_{15,6})\mathbf{a}_e + (b_{16,12} - b_{15,12}b_{16,6}/b_{15,6})\mathbf{a}_f + (b_{16,13} - b_{15,13}b_{16,6}/b_{15,6})\mathbf{a}_g + (b_{16,14} - b_{15,14}b_{16,6}/b_{15,6})\mathbf{a}_h$
60. $x^0 = x^0 + (a_{15} - a_{14})h$
61. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_i\right]\right]$
62. $\mathbf{a}_i = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_i, \mathbf{p}, \xi\right)$
63. $x^0 = x^0 + (a_{16} - a_{15})h$
64. $\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}_j\right]\right]$
65. $\mathbf{a}_j = \mathcal{N}\left(\mathbf{x}, \mathbf{a}_j, \mathbf{p}, \xi\right)$

66. $\mathbf{a}_{Int} = \mathbf{a}$

67. $\mathbf{a} = \mathbf{a} + c_1\mathbf{a}_a + c_8\mathbf{a}_b + c_9\mathbf{a}_c + c_{10}\mathbf{a}_d + c_{11}\mathbf{a}_e + c_{12}\mathbf{a}_f + c_{13}\mathbf{a}_g + c_{14}\mathbf{a}_h + c_{15}\mathbf{a}_i + c_{16}\mathbf{a}_j$

68. $\mathbf{a}_a = \mathbf{a}_{Int} + c_1^*\mathbf{a}_a + c_8^*\mathbf{a}_b + c_9^*\mathbf{a}_c + c_{10}^*\mathbf{a}_d + c_{11}^*\mathbf{a}_e + c_{12}^*\mathbf{a}_f + c_{13}^*\mathbf{a}_g + c_{14}^*\mathbf{a}_h + c_{15}^*\mathbf{a}_i + c_{16}^*\mathbf{a}_j$

This method requires 11 extra copies of the field. An additional copy is required for the adaptive time-step (compared to the non adaptive ARK9EX) since the approximate error in the step must be calculated. This is achieved using the field stored in $\mathbf{a}_a$ which holds the 8th order solution, analogous to the $\mathbf{a}^*$ in the ARK45 method. Once again we should note the adaptive step implemented in this method is stochastically safe, making it the best performing solver for stochastic problems.

### 6.4.8 Interaction Picture Methods

Interaction picture methods can only be applied to equations of the general form:

$$\frac{\partial}{\partial x^0}a^i(x^0,\mathbf{x}_\perp) = \mathcal{F}^{-1}\left[\mathcal{L}^i(x^0,\mathbf{k}_\perp)\mathcal{F}[a^i(\mathbf{x})]\right] + \mathcal{N}^i\left(\mathbf{x},\mathbf{a}(\mathbf{x}),\xi(\mathbf{x})\right). \tag{6.37}$$

Although it is possible to define interaction picture algorithms in which the linear operators may have the general matrix form $\mathcal{L}^{ij}$, here they are restricted to the diagonal form $\mathcal{L}^i$. Non-diagonal operators require eigen-vector-value decomposition each time step and at each lattice point, which becomes computationally expensive. These $\mathcal{L}^i$ operators may still have coefficients that depend on the the propagation dimension and an the Fourier space coordinates $\mathbf{k}_\perp$.

The evolution of the field $\mathbf{a}$ is carried out in both normal space *and* Fourier space in alternating steps, hence this method is also known as a *Split-Step* method, or a *Split-Operator* method. The exact implementation depends on the main integration method employed, as is detailed in the following Sections, 6.4.9 and 6.4.10.

As a general rule the interaction picture algorithms are faster and more stable than their Explicit picture counterparts, but they are also more restrictive with regard to the allowable linear operators.

### 6.4.9 The Semi-Implicit method in the Interaction Picture

The original derivation of this method was performed by Drummond [6], and we repeat it here since it is the most efficient and stable method for solving stochastic PDEs. Consider now expressing the field $a$ as the transform:

$$a^i(x^0,\mathbf{x}_\perp) = e^{\left(x^0-z\right)\mathcal{L}^i\left(x^0,\mathbf{k}_\perp\right)}\left[b^i(x^0,\mathbf{x}_\perp)\right]. \tag{6.38}$$

The time derivative of $\mathbf{a}$ then becomes:

$$\frac{\partial}{\partial x^0}a^i(x^0,\mathbf{x}_\perp) = \mathcal{L}^i\left(x^0,\mathbf{k}_\perp\right)\left[a^i\right] + e^{\left(x^0-z\right)f^i(x^0,\mathbf{k}_\perp)\mathcal{L}^i}\left[\dot{b}^i\right], \tag{6.39}$$

since the operators $\mathcal{L}$ are linear. Equating this result with Equation (6.37) we get:

$$\dot{b}^i(x^0, \mathbf{x}_\perp) = e^{-\left(x^0 - z\right)\mathcal{L}^i\left(x^0, \mathbf{k}_\perp\right)} \left[\mathcal{N}^i\left(\mathbf{x}, \mathbf{a}, \xi\right)\right]. \tag{6.40}$$

Now solving for $\mathbf{b}$ using the semi-implicit method yields:

$$b^i\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp\right) = b^i\left(x^0, \mathbf{x}_\perp\right) \tag{6.41}$$

$$+ \frac{1}{2}\, h\, e^{-\left(x^0 + \frac{h}{2} - z\right)\mathcal{L}^i\left(x^0, \mathbf{k}_\perp\right)} \left[\mathcal{N}^i\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp, \mathbf{a}\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp\right), \xi\right)\right]. \tag{6.42}$$

Finally, if we define $z = x^0 + \frac{h}{2}$ then $\mathbf{a}$ and $\mathbf{b}$ become identical at the middle of the time step, and we obtain

$$b^i\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp\right) = b^i\left(x^0, \mathbf{x}_\perp\right) + \frac{h}{2}\mathcal{N}^i\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp, \mathbf{b}\left(x^0 + \frac{h}{2}, \mathbf{x}_\perp\right), \xi\right). \tag{6.43}$$

Hence the $\mathbf{a}$ and $\mathbf{b}$ fields are transformed from one to the other at the beginning and end of the time step using the linear operators, and the $\mathbf{b}$ field is evolved over the time step using the nonlinear operators and the semi-implicit algorithm. Thus the algorithm becomes quite simply:

1. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}$
2. Calculate $\xi$ if required.
3. $x^0 = x^0 + \frac{1}{2}h$
4. For each point in space do:
   (a) $\mathbf{a_I} = \mathbf{a}$
   (b) For N-1 iterations do:
      i. $\mathbf{a} = \mathbf{a}_I + \frac{1}{2}h\,\mathcal{N}(\mathbf{x}, \mathbf{a}, \xi)$
   (c) $\mathbf{a} = \mathbf{a}_I + h\,\mathcal{N}(\mathbf{x}, \mathbf{a}, \xi)$
5. $x^0 = x^0 + \frac{1}{2}h$
6. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}$

The first and the last steps are easily performed with the field in Fourier space since the linear operators reduce to $k_\perp$ multipliers. Note that the dependence of the linear operators $\mathcal{L}^i(x^0, \mathbf{k}_\perp)$ on the propagation dimension ought to be weak in comparison to the size of the time step. If they do vary with time then the exponentials will have to be calculated at every time step, which can be computationally expensive with some CPU architectures. Otherwise the exponentials may be pre-calculated and tabulated for reference.

Further advantages of this method are that the forward and backward Fourier transforms may be performed *in place* (i.e. within the current memory block), as can the point-by-point multiplication of the field by the exponentials. Thus no extra copies of the field are required.

### 6.4.10 The Fourth Order Runge-Kutta Method in the Interaction Picture

One way of reducing the memory overhead on the RK4 algorithm is to move into an interaction picture, exactly as was performed in the split-step semi-implicit method above. This method was derived by Rob Ballagh's BEC group at the University of Otago. It is described in detail in the PhD thesis of B.M. Caradoc-Davies [7].

We will present only an optimised recipe for implementing this algorithm:

1. Calculate $\xi$ if required.
2. $\mathbf{a}_K = \mathbf{a}$
3. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$
4. $\mathbf{a}_I = \mathbf{a}$
5. $\mathbf{a}_K = h\,\mathcal{N}(\mathbf{x}, \mathbf{a}_K, \xi)$
6. $\mathbf{a}_K = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_K$
7. $\mathbf{a} = \mathbf{a} + \frac{1}{6}\mathbf{a}_K$
8. $x^0 = x^0 + \frac{1}{2}h$
9. $\mathbf{a}_K = \mathbf{a}_I + \frac{1}{2}\mathbf{a}_K$
10. $\mathbf{a}_K = h\,\mathcal{N}(\mathbf{x}, \mathbf{a}_K, \xi)$
11. $\mathbf{a} = \mathbf{a} + \frac{1}{3}\mathbf{a}_K$
12. $\mathbf{a}_K = \mathbf{a}_I + \frac{1}{2}\mathbf{a}_K$
13. $\mathbf{a}_K = h\,\mathcal{N}(\mathbf{x}, \mathbf{a}_K, \xi)$
14. $\mathbf{a} = \mathbf{a} + \frac{1}{3}\mathbf{a}_K$
15. $x^0 = x^0 + \frac{1}{2}h$
16. $\mathbf{a}_K = \mathbf{a}_I + \mathbf{a}_K$
17. $\mathbf{a}_K = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_K$
18. $\mathbf{a}_K = h\,\mathcal{N}(\mathbf{x}, \mathbf{a}_K, \xi)$
19. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$
20. $\mathbf{a} = \mathbf{a} + \frac{1}{6}\mathbf{a}_K$

Similarly to the semi-implicit algorithm in the interaction picture, the first and the last steps are easily performed with the field in Fourier space since the linear operators reduce to $k_\perp$ multipliers. And again note that the dependence of the linear operators $\mathcal{L}^i(x^0, \mathbf{k}_\perp)$ on the propagation dimension ought to be weak in comparison to the size of the time step.

This algorithm has a similar memory overhead as its Explicit picture counterpart, but it no longer needs the memory for the $\mathbf{p}$ derivatives vector, nor does it have to do the work to calculate this vector.

The interaction picture method relies on the derivative Operator being independent of the propagation dimension, otherwise it cannot function as a fourth order algorithm.

### 6.4.11 The Fourth/ Fifth Order adaptive Runge-Kutta Method in the Interaction Picture

In the Fourth/ Fifth Order adaptive Runge-Kutta algorithm it seems we would require too many Fourier transforms for this to be efficient. Following [8] the function can however primarily be evolved in Fourier space and transformed into normal space for calculation of the $\mathcal{N}(\mathbf{x}, \mathbf{a}_i, \xi)$ only. The use of the interaction picture does not allow a reduction of the computational effort to the same extent as in the RK4IP algorithm, but this method can still be vastly superior over the adaptive explicit picture method as it allows larger step sizes for equations containing certain derivative operators. In the following the vectors $\mathbf{a}$, $\mathbf{a}^*$ are supposed to be initially in Fourier space. Note that for the calculation of the $\mathcal{N}()$ these are transformed into normal space.

1. Calculate $\xi$ if required.
2. $\mathbf{a}_K = \mathbf{a}$
3. $\mathbf{a} = e^{a_2 h \mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}$
4. $\mathbf{a}_I = \mathbf{a}$
5. $\mathbf{a}^* = \mathbf{a}$
6. $\mathbf{a}_K = h\, \mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_K\right], \xi\right)\right]$
7. $\mathbf{a}_K = e^{-a_2 h \mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_K$
8. $\mathbf{a} = \mathbf{a} + c_1 \mathbf{a}_K$
9. $\mathbf{a}^* = \mathbf{a}^* + c_1^* \mathbf{a}_K$
10. $x^0 = x^0 + a_2 h$
11. $\mathbf{a}_K = \mathbf{a}_I + b_{21} \mathbf{a}_K$
12. $\mathbf{a}_K = h\, \mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_K\right], \xi\right)\right]$
13. $\mathbf{a}_J = (1 - b_{31}/c_1)\mathbf{a}_I + b_{31}/c_1 \mathbf{a} + b_{32} \mathbf{a}_K$

14. $x^0 = x^0 + (a_3 - a_2)h$
    (Note: $c_2 = c_2^* = 0$)
15. $\mathbf{a}_J = e^{(a_3-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_J$
16. $\mathbf{a}_J = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_J\right], \xi\right)\right]$
17. $\mathbf{a}_J = e^{-(a_3-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_J$
18. $\mathbf{a}_L = (1 - b_{41}/c_1)\mathbf{a}_I + b_{41}/c_1\mathbf{a} + b_{42}\mathbf{a}_K + b_{43}\mathbf{a}_J$
19. $\mathbf{a} = \mathbf{a} + c_3\mathbf{a}_J$
20. $\mathbf{a}^* = \mathbf{a}^* + c_3^*\mathbf{a}_J$
21. $x^0 = x^0 + (a_4 - a_3)h$
22. $\mathbf{a}_L = e^{(a_4-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
23. $\mathbf{a}_L = h\,\mathcal{F}\left[\mathcal{N}\left(\mathbf{x}, \mathcal{F}^{-1}\left[\mathbf{a}_L\right], \xi\right)\right]$
24. $\mathbf{a}_L = e^{-(a_4-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
25. $\mathbf{a} = \mathbf{a} + c_4\mathbf{a}_L$
26. $\mathbf{a}^* = \mathbf{a}^* + c_4^*\mathbf{a}_L$
27. $\mathbf{a}_L = (1 - b_{51}/c_1)\mathbf{a}_I + b_{51}/c_1\mathbf{a} + b_{52}\mathbf{a}_K + (b_{53} - b_{51}c_3/c_1)\mathbf{a}_J + (b_{54} - b_{51}c_4/c_1)\mathbf{a}_L$
28. $x^0 = x^0 + (a_5 - a_4)h$
29. $\mathbf{a}_L = e^{(a_5-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
30. $\mathbf{a}_L = h\,\mathcal{F}\left[\mathcal{N}\left(\mathbf{x}, \mathcal{F}^{-1}\left[\mathbf{a}_L\right], \xi\right)\right]$
31. $\mathbf{a}_L = e^{-(a_5-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
32. $\mathbf{a}^* = \mathbf{a}^* + c_5^*\mathbf{a}_L$
    (Note: $c_5 = 0$)
33. $\mathbf{a}_L = f_1\mathbf{a}_I + f_2\mathbf{a}_K + f_3\mathbf{a}_J + f_4\mathbf{a} + f_5\mathbf{a}_L + f_6\mathbf{a}^*$
34. $x^0 = x^0 + (a_6 - a_5)h$
35. $\mathbf{a}_L = e^{(a_6-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
36. $\mathbf{a}_L = h\,\mathcal{F}\left[\mathcal{N}\left(\mathbf{x}, \mathcal{F}^{-1}\left[\mathbf{a}_L\right], \xi\right)\right]$

37. $\mathbf{a}_L = e^{-(a_6-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_L$
38. $\mathbf{a} = \mathbf{a} + c_6\mathbf{a}_J$
39. $\mathbf{a}^* = \mathbf{a}^* + c_6^*\mathbf{a}_J$
40. $x^0 = x^0 + (a_6 - a_5)h$
41. $\mathbf{a} = e^{(1-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$
42. $\mathbf{a}^* = e^{(1-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}^*$

The $f_i$ coefficients are listed in section 6.4.5. Unlike its fixed step counterpart (RK4IP), this algorithm requires significant memory for the fastest calculation of the Fourier space propagation, as the arguments of the exponentials contain different factors of $a_i - a_2$ for each application. If this turns out to be a problem, XMDS offers the possibility to sacrifice speed for less memory consumption by the use of an appropriate flag.

As the two computed solutions are of fourth and fifth order *only* if the derivative operators are independent of the propagation dimension, this algorithm cannot be used at all for problems where this is not the case.

### 6.4.12 The Ninth Order Runge-Kutta Method in the Interaction Picture

The ninth order Runge-Kutta does require a large amount of Fourier transforms, since there is no temporal Symmetry in the algorithm. However the improvement in convergence may offset the extra time required, particularly in stochastic problems.

1. Calculate $\xi$ if required.
2. $x^0 = x^0 + a_1 h$
3. $\mathbf{a}_a = \mathbf{a}$
4. $\mathbf{a} = e^{h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$
5. $\mathbf{a}_a = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_a\right], \xi\right)\right]$
6. $\mathbf{a}_a = e^{h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_a$
7. $x^0 = x^0 + (a_2 - a_1)h$
8. $\mathbf{a}_b = \mathbf{a} + b_{2,1}\mathbf{a}_a$
9. $\mathbf{a}_b = e^{-(1-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
10. $\mathbf{a}_b = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_b\right], \xi\right)\right]$

11. $\mathbf{a}_b = e^{(1-a_2)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
12. $x^0 = x^0 + (a_3 - a_2)h$
13. $\mathbf{a}_c = \mathbf{a} + b_{3,1}\mathbf{a}_a + b_{3,2}\mathbf{a}_b$
14. $\mathbf{a}_c = e^{-(1-a_3)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
15. $\mathbf{a}_c = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_c\right], \xi\right)\right]$
16. $\mathbf{a}_c = e^{(1-a_3)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
17. $x^0 = x^0 + (a_4 - a_3)h$
18. $\mathbf{a}_d = \mathbf{a} + b_{4,1}\mathbf{a}_a + b_{4,2}\mathbf{a}_b + b_{4,3}\mathbf{a}_c$
19. $\mathbf{a}_d = e^{-(1-a_4)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
20. $\mathbf{a}_d = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_d\right], \xi\right)\right]$
21. $\mathbf{a}_d = e^{(1-a_4)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
22. $x^0 = x^0 + (a_5 - a_4)h$
23. $\mathbf{a}_e = \mathbf{a} + b_{5,1}\mathbf{a}_a + b_{5,2}\mathbf{a}_b + b_{5,3}\mathbf{a}_c + b_{5,4}\mathbf{a}_d$
24. $\mathbf{a}_e = e^{-(1-a_5)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
25. $\mathbf{a}_e = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_e\right], \xi\right)\right]$
26. $\mathbf{a}_e = e^{(1-a_5)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
27. $x^0 = x^0 + (a_6 - a_5)h$
28. $\mathbf{a}_i = \mathbf{a} + b_{6,1}\mathbf{a}_a + b_{6,2}\mathbf{a}_b + b_{6,3}\mathbf{a}_c + b_{6,4}\mathbf{a}_d + b_{6,5}\mathbf{a}_e$
29. $\mathbf{a}_i = e^{-(1-a_6)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_i$
30. $\mathbf{a}_i = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_i\right], \xi\right)\right]$
31. $\mathbf{a}_i = e^{(1-a_6)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_i$
32. $x^0 = x^0 + (a_7 - a_6)h$
33. $\mathbf{a}_j = \mathbf{a} + b_{7,1}\mathbf{a}_a + b_{7,2}\mathbf{a}_b + b_{7,3}\mathbf{a}_c + b_{7,4}\mathbf{a}_d + b_{7,5}\mathbf{a}_e + b_{7,6}\mathbf{a}_i$
34. $\mathbf{a}_j = e^{-(1-a_7)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_j$
35. $\mathbf{a}_j = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_j\right], \xi\right)\right]$

36. $\mathbf{a}_j = e^{(1-a_7)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_j$
37. $x^0 = x^0 + (a_8 - a_7)h$
38. $\mathbf{a}_b = \mathbf{a} + b_{8,1}\mathbf{a}_a + b_{8,6}\mathbf{a}_i + b_{8,7}\mathbf{a}_j$
39. $\mathbf{a}_b = e^{-(1-a_8)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
40. $\mathbf{a}_b = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_b\right], \xi\right)\right]$
41. $\mathbf{a}_b = e^{(1-a_8)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
42. $x^0 = x^0 + (a_9 - a_8)h$
43. $\mathbf{a}_c = \mathbf{a} + b_{9,1}\mathbf{a}_a + b_{9,6}\mathbf{a}_i + b_{9,7}\mathbf{a}_j + b_{9,8}\mathbf{a}_b$
44. $\mathbf{a}_c = e^{-(1-a_9)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
45. $\mathbf{a}_c = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_c\right], \xi\right)\right]$
46. $\mathbf{a}_c = e^{(1-a_9)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
47. $x^0 = x^0 + (a_{10} - a_9))h$
48. $\mathbf{a}_d = \mathbf{a} + b_{10,1}\mathbf{a}_a + b_{10,6}\mathbf{a}_i + b_{10,7}\mathbf{a}_j + b_{10,8}\mathbf{a}_b + b_{10,9}\mathbf{a}_c$
49. $\mathbf{a}_d = e^{-(1-a_{10})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
50. $\mathbf{a}_d = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_d\right], \xi\right)\right]$
51. $\mathbf{a}_d = e^{(1-a_{10})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
52. $x^0 = x^0 + (a_{11} - a_{10})h$
53. $\mathbf{a}_e = \mathbf{a} + b_{11,1}\mathbf{a}_a + b_{11,6}\mathbf{a}_i + b_{11,7}\mathbf{a}_j + b_{11,8}\mathbf{a}_b + b_{11,9}\mathbf{a}_c + b_{11,10}\mathbf{a}_d$
54. $\mathbf{a}_e = e^{-(1-a_{11})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
55. $\mathbf{a}_e = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_e\right], \xi\right)\right]$
56. $\mathbf{a}_e = e^{(1-a_{11})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
57. $x^0 = x^0 + (a_{12} - a_{11})h$
58. $\mathbf{a}_f = \mathbf{a} + b_{12,1}\mathbf{a}_a + b_{12,6}\mathbf{a}_i + b_{12,7}\mathbf{a}_j + b_{12,8}\mathbf{a}_b + b_{12,9}\mathbf{a}_c + b_{12,10}\mathbf{a}_d + b_{12,11}\mathbf{a}_e$
59. $\mathbf{a}_f = e^{-(1-a_{12})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_f$
60. $\mathbf{a}_f = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_f\right], \xi\right)\right]$

61. $\mathbf{a}_f = e^{(1-a_{12})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_f$

62. $x^0 = x^0 + (a_{13} - a_{12})h$

63. $\mathbf{a}_g = \mathbf{a} + b_{13,1}\mathbf{a}_a + b_{13,6}\mathbf{a}_i + b_{13,7}\mathbf{a}_j + b_{13,8}\mathbf{a}_b + b_{13,9}\mathbf{a}_c + b_{13,10}\mathbf{a}_d + b_{13,11}\mathbf{a}_e + b_{13,12}\mathbf{a}_f$

64. $\mathbf{a}_g = e^{-(1-a_{13})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_g$

65. $\mathbf{a}_g = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_g\right], \xi\right)\right]$

66. $\mathbf{a}_g = e^{(1-a_{13})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_g$

67. $x^0 = x^0 + (a_{14} - a_{13})h$

68. $\mathbf{a}_h = \mathbf{a} + b_{14,1}\mathbf{a}_a + b_{14,6}\mathbf{a}_i + b_{14,7}\mathbf{a}_j + b_{14,8}\mathbf{a}_b + b_{14,9}\mathbf{a}_c + b_{14,10}\mathbf{a}_d + b_{14,11}\mathbf{a}_e + b_{14,12}\mathbf{a}_f + b_{14,13}\mathbf{a}_g$

69. $\mathbf{a}_h = e^{-(1-a_{14})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_h$

70. $\mathbf{a}_h = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_h\right], \xi\right)\right]$

71. $\mathbf{a}_h = e^{(1-a_{14})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_h$

72. $\mathbf{a}_i = \mathbf{a} + b_{15,1}\mathbf{a}_a + b_{15,6}\mathbf{a}_i + b_{15,7}\mathbf{a}_j + b_{15,8}\mathbf{a}_b + b_{15,9}\mathbf{a}_c + b_{15,10}\mathbf{a}_d + b_{15,11}\mathbf{a}_e + b_{15,12}\mathbf{a}_f + b_{15,13}\mathbf{a}_g + b_{15,14}\mathbf{a}_h$

73. $\mathbf{a}_j = (1-b_{16,6}/b_{15,6})\mathbf{a} + (b_{16,1} - b_{15,1}b_{16,6}/b_{15,6})\mathbf{a}_a + (b_{16,6}/b_{15,6})\mathbf{a}_i + (b_{16,7} - b_{15,7}b_{16,6}/b_{15,6})\mathbf{a}_j + (b_{16,8} - b_{16,8}b_{16,6}/b_{15,6})\mathbf{a}_b + (b_{16,9} - b_{15,9}b_{16,6}/b_{15,6})\mathbf{a}_c + (b_{16,10} - b_{15,10}b_{16,6}/b_{15,6})\mathbf{a}_d + (b_{16,11} - b_{15,11}b_{16,6}/b_{15,6})\mathbf{a}_e + (b_{16,12} - b_{15,12}b_{16,6}/b_{15,6})\mathbf{a}_f + (b_{16,13} - b_{15,13}b_{16,6}/b_{15,6})\mathbf{a}_g + (b_{16,14} - b_{15,14}b_{16,6}/b_{15,6})\mathbf{a}_h$

74. $x^0 = x^0 + (a_{15} - a_{14})h$

75. $\mathbf{a}_i = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_i\right], \xi\right)\right]$

76. $x^0 = x^0 + (a_{16} - a_{15})h$

77. $\mathbf{a}_j = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_j\right], \xi\right)\right]$

78. $\mathbf{a} = \mathbf{a} + c_1\mathbf{a}_a + c_8\mathbf{a}_b + c_9\mathbf{a}_c + c_{10}\mathbf{a}_d + c_{11}\mathbf{a}_e + c_{12}\mathbf{a}_f + c_{13}\mathbf{a}_g + c_{14}\mathbf{a}_h + c_{15}\mathbf{a}_i + c_{16}\mathbf{a}_j$

The method requires 10 copies of the field like its explicit picture counterpart. Note no rotations are required for steps 15 and 16 since they both occur at the end of the time step.

### 6.4.13 The Eighth / Ninth Order adaptive Runge-Kutta Method in the Interaction Picture

The Eighth/Ninth order Runge-Kutta also requires a large number of Fourier transforms since there is no temporal symmetry in the algorithm. However, the improvement in convergence may offset the extra time required, particularly in stochastic problems.

1. Calculate $\xi$ if required.
2. $x^0 = x^0 + a_1 h$
3. $\mathbf{a}_a = \mathbf{a}$
4. $\mathbf{a} = e^{h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}$
5. $\mathbf{a}_a = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_a\right], \xi\right)\right]$
6. $\mathbf{a}_a = e^{h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_a$
7. $x^0 = x^0 + (a_2 - a_1)h$
8. $\mathbf{a}_b = \mathbf{a} + b_{2,1}\mathbf{a}_a$
9. $\mathbf{a}_b = e^{-(1-a_2)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
10. $\mathbf{a}_b = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_b\right], \xi\right)\right]$
11. $\mathbf{a}_b = e^{(1-a_2)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
12. $x^0 = x^0 + (a_3 - a_2)h$
13. $\mathbf{a}_c = \mathbf{a} + b_{3,1}\mathbf{a}_a + b_{3,2}\mathbf{a}_b$
14. $\mathbf{a}_c = e^{-(1-a_3)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
15. $\mathbf{a}_c = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_c\right], \xi\right)\right]$
16. $\mathbf{a}_c = e^{(1-a_3)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
17. $x^0 = x^0 + (a_4 - a_3)h$
18. $\mathbf{a}_d = \mathbf{a} + b_{4,1}\mathbf{a}_a + b_{4,2}\mathbf{a}_b + b_{4,3}\mathbf{a}_c$
19. $\mathbf{a}_d = e^{-(1-a_4)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
20. $\mathbf{a}_d = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_d\right], \xi\right)\right]$
21. $\mathbf{a}_d = e^{(1-a_4)h\mathcal{L}\left(x^0, \mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
22. $x^0 = x^0 + (a_5 - a_4)h$

23. $\mathbf{a}_e = \mathbf{a} + b_{5,1}\mathbf{a}_a + b_{5,2}\mathbf{a}_b + b_{5,3}\mathbf{a}_c + b_{5,4}\mathbf{a}_d$
24. $\mathbf{a}_e = e^{-(1-a_5)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
25. $\mathbf{a}_e = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_e\right], \xi\right)\right]$
26. $\mathbf{a}_e = e^{(1-a_5)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
27. $x^0 = x^0 + (a_6 - a_5)h$
28. $\mathbf{a}_i = \mathbf{a} + b_{6,1}\mathbf{a}_a + b_{6,2}\mathbf{a}_b + b_{6,3}\mathbf{a}_c + b_{6,4}\mathbf{a}_d + b_{6,5}\mathbf{a}_e$
29. $\mathbf{a}_i = e^{-(1-a_6)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_i$
30. $\mathbf{a}_i = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_i\right], \xi\right)\right]$
31. $\mathbf{a}_i = e^{(1-a_6)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_i$
32. $x^0 = x^0 + (a_7 - a_6)h$
33. $\mathbf{a}_j = \mathbf{a} + b_{7,1}\mathbf{a}_a + b_{7,2}\mathbf{a}_b + b_{7,3}\mathbf{a}_c + b_{7,4}\mathbf{a}_d + b_{7,5}\mathbf{a}_e + b_{7,6}\mathbf{a}_i$
34. $\mathbf{a}_j = e^{-(1-a_7)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_j$
35. $\mathbf{a}_j = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_j\right], \xi\right)\right]$
36. $\mathbf{a}_j = e^{(1-a_7)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_j$
37. $x^0 = x^0 + (a_8 - a_7)h$
38. $\mathbf{a}_b = \mathbf{a} + b_{8,1}\mathbf{a}_a + b_{8,6}\mathbf{a}_i + b_{8,7}\mathbf{a}_j$
39. $\mathbf{a}_b = e^{-(1-a_8)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
40. $\mathbf{a}_b = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_b\right], \xi\right)\right]$
41. $\mathbf{a}_b = e^{(1-a_8)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_b$
42. $x^0 = x^0 + (a_9 - a_8)h$
43. $\mathbf{a}_c = \mathbf{a} + b_{9,1}\mathbf{a}_a + b_{9,6}\mathbf{a}_i + b_{9,7}\mathbf{a}_j + b_{9,8}\mathbf{a}_b$
44. $\mathbf{a}_c = e^{-(1-a_9)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
45. $\mathbf{a}_c = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_c\right], \xi\right)\right]$
46. $\mathbf{a}_c = e^{(1-a_9)h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_c$
47. $x^0 = x^0 + (a_{10} - a_9))h$

48. $\mathbf{a}_d = \mathbf{a} + b_{10,1}\mathbf{a}_a + b_{10,6}\mathbf{a}_i + b_{10,7}\mathbf{a}_j + b_{10,8}\mathbf{a}_b + b_{10,9}\mathbf{a}_c$
49. $\mathbf{a}_d = e^{-(1-a_{10})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
50. $\mathbf{a}_d = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_d\right], \xi\right)\right]$
51. $\mathbf{a}_d = e^{(1-a_{10})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_d$
52. $x^0 = x^0 + (a_{11} - a_{10})h$
53. $\mathbf{a}_e = \mathbf{a} + b_{11,1}\mathbf{a}_a + b_{11,6}\mathbf{a}_i + b_{11,7}\mathbf{a}_j + b_{11,8}\mathbf{a}_b + b_{11,9}\mathbf{a}_c + b_{11,10}\mathbf{a}_d$
54. $\mathbf{a}_e = e^{-(1-a_{11})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
55. $\mathbf{a}_e = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_e\right], \xi\right)\right]$
56. $\mathbf{a}_e = e^{(1-a_{11})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_e$
57. $x^0 = x^0 + (a_{12} - a_{11})h$
58. $\mathbf{a}_f = \mathbf{a} + b_{12,1}\mathbf{a}_a + b_{12,6}\mathbf{a}_i + b_{12,7}\mathbf{a}_j + b_{12,8}\mathbf{a}_b + b_{12,9}\mathbf{a}_c + b_{12,10}\mathbf{a}_d + b_{12,11}\mathbf{a}_e$
59. $\mathbf{a}_f = e^{-(1-a_{12})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_f$
60. $\mathbf{a}_f = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_f\right], \xi\right)\right]$
61. $\mathbf{a}_f = e^{(1-a_{12})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_f$
62. $x^0 = x^0 + (a_{13} - a_{12})h$
63. $\mathbf{a}_g = \mathbf{a} + b_{13,1}\mathbf{a}_a + b_{13,6}\mathbf{a}_i + b_{13,7}\mathbf{a}_j + b_{13,8}\mathbf{a}_b + b_{13,9}\mathbf{a}_c + b_{13,10}\mathbf{a}_d + b_{13,11}\mathbf{a}_e + b_{13,12}\mathbf{a}_f$
64. $\mathbf{a}_g = e^{-(1-a_{13})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_g$
65. $\mathbf{a}_g = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_g\right], \xi\right)\right]$
66. $\mathbf{a}_g = e^{(1-a_{13})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_g$
67. $x^0 = x^0 + (a_{14} - a_{13})h$
68. $\mathbf{a}_h = \mathbf{a} + b_{14,1}\mathbf{a}_a + b_{14,6}\mathbf{a}_i + b_{14,7}\mathbf{a}_j + b_{14,8}\mathbf{a}_b + b_{14,9}\mathbf{a}_c + b_{14,10}\mathbf{a}_d + b_{14,11}\mathbf{a}_e + b_{14,12}\mathbf{a}_f + b_{14,13}\mathbf{a}_g$
69. $\mathbf{a}_h = e^{-(1-a_{14})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_h$
70. $\mathbf{a}_h = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_h\right], \xi\right)\right]$
71. $\mathbf{a}_h = e^{(1-a_{14})h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}_h$
72. $\mathbf{a}_i = \mathbf{a} + b_{15,1}\mathbf{a}_a + b_{15,6}\mathbf{a}_i + b_{15,7}\mathbf{a}_j + b_{15,8}\mathbf{a}_b + b_{15,9}\mathbf{a}_c + b_{15,10}\mathbf{a}_d + b_{15,11}\mathbf{a}_e + b_{15,12}\mathbf{a}_f + b_{15,13}\mathbf{a}_g + b_{15,14}\mathbf{a}_h$

73. $\mathbf{a}_j = (1-b_{16,6}/b_{15,6})\mathbf{a}+(b_{16,1}-b_{15,1}b_{16,6}/b_{15,6})\mathbf{a}_a+(b_{16,6}/b_{15,6})\mathbf{a}_i+(b_{16,7}-b_{15,7}b_{16,6}/b_{15,6})\mathbf{a}_j+(b_{16,8}-b_{16,8}b_{16,6}/b_{15,6})\mathbf{a}_b+(b_{16,9}-b_{15,9}b_{16,6}/b_{15,6})\mathbf{a}_c+(b_{16,10}-b_{15,10}b_{16,6}/b_{15,6})\mathbf{a}_d+(b_{16,11}-b_{15,11}b_{16,6}/b_{15,6})\mathbf{a}_e + (b_{16,12} - b_{15,12}b_{16,6}/b_{15,6})\mathbf{a}_f + (b_{16,13} - b_{15,13}b_{16,6}/b_{15,6})\mathbf{a}_g + (b_{16,14} - b_{15,14}b_{16,6}/b_{15,6})\mathbf{a}_h$

74. $x^0 = x^0 + (a_{15} - a_{14})h$

75. $\mathbf{a}_i = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_i\right], \xi\right)\right]$

76. $x^0 = x^0 + (a_{16} - a_{15})h$

77. $\mathbf{a}_j = h\,\mathcal{F}\left[\mathcal{N}\left(x^0, \mathcal{F}^{-1}\left[\mathbf{a}_j\right], \xi\right)\right]$

78. $\mathbf{a}_{Int} = \mathbf{a}$

79. $\mathbf{a} = \mathbf{a} + c_1\mathbf{a}_a + c_8\mathbf{a}_b + c_9\mathbf{a}_c + c_{10}\mathbf{a}_d + c_{11}\mathbf{a}_e + c_{12}\mathbf{a}_f + c_{13}\mathbf{a}_g + c_{14}\mathbf{a}_h + c_{15}\mathbf{a}_i + c_{16}\mathbf{a}_j$

80. $\mathbf{a}_a = \mathbf{a}_{Int} + c_1^*\mathbf{a}_a + c_8^*\mathbf{a}_b + c_9^*\mathbf{a}_c + c_{10}^*\mathbf{a}_d + c_{11}^*\mathbf{a}_e + c_{12}^*\mathbf{a}_f + c_{13}^*\mathbf{a}_g + c_{14}^*\mathbf{a}_h + c_{15}^*\mathbf{a}_i + c_{16}^*\mathbf{a}_j$

The method requires 11 copies of the field and has a variable time step which is stochastically safe like its explicit picture counterpart. Once again no rotations are required for steps 15 and 16 since they both occur at the end of the time step.

### 6.4.14 Adding a Cross Vector

The nonlinear functionals of Equations (6.35) and (6.37) may also include variables that are a function of the same transverse space, but are which are governed by ODEs in one of the transverse dimensions. These components need never be transformed to Fourier space, and so ought to be separated from the main vector components for efficiency reasons. In Equation (8.3) they are referred to as the vector $\mathbf{b}$, which propagates in dimension $x^c$. This vector has a definite value once the main vector $\mathbf{a}$ has a definite value. This means that for most integration algorithms a current value for the cross vector may be calculated immediately prior to calculating the $\mathcal{N}$ functionals.

In the semi-implicit method in the explicit picture the cross vector $\mathbf{b}$ is calculated using the semi-implicit method and using linear interpolation in the $x^c$ dimension to create the mid-lattice point values for $\mathbf{a}$ that are required.

Similarly, with the RK4 method, in both the explicit and interaction pictures, the calculation of $\mathbf{b}$ is performed using the RK4 method, and again using linear interpolation in the $x^c$ dimension to create the mid-lattice point values for $\mathbf{a}$ that are required. Note that this use of simple linear interpolation causes a loss of order when implementing a higher order method such as the RK4, but generally the transverse lattice is sufficiently fine for this to be negligible.

However, as mentioned earlier the semi-implicit method in the interaction picture need only sweep through the memory space of main field vector once with each nonlinear step. Therefore it would be desirable to use an algorithm that can calculate the cross vector simultaneously in the same sweep. The semi-implicit method in the interaction picture does

this by employing the standard semi-implicit algorithm in the propagation direction *simultaneously* with another semi-implicit algorithm in the transverse propagation dimension. In pseudo-code it looks like this:

1. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$
2. Calculate $\xi$ if required.
3. $x^0 = x^0 + \frac{1}{2}h$
4. For each point in space do:
   (a) $\mathbf{a}_I = \mathbf{a}$
   (b) $\mathbf{b}_I = \mathbf{b}$
   (c) For N-1 iterations do:
      i. $\mathbf{a} = \mathbf{a}_I + \frac{1}{2}h\,\mathcal{N}(\mathbf{x},\mathbf{a},\mathbf{b},\xi)$
      ii. $\mathbf{b} = \mathbf{b}_I + \frac{1}{2}\Delta x^c\,\mathcal{H}(\mathbf{x},\mathbf{a},\mathbf{b})$
   (d) $\mathbf{a} = \mathbf{a}_I + h\,\mathcal{N}(\mathbf{x},\mathbf{a},\mathbf{b},\xi)$
   (e) $\mathbf{b} = \mathbf{b}_I + \frac{1}{2}\Delta x^c\,\mathcal{H}(\mathbf{x},\mathbf{a},\mathbf{b})$
   (f) $\mathbf{b}(x^c + \Delta x^c) = \mathbf{b}_I + \Delta x^c\,\mathcal{H}(\mathbf{x},\mathbf{a},\mathbf{b})$
5. $x^0 = x^0 + \frac{1}{2}h$
6. $\mathbf{a} = e^{\frac{1}{2}h\mathcal{L}\left(x^0,\mathbf{k}_\perp\right)} \cdot \mathbf{a}$

This routine assumes that the cross vector $b$ was already initialised at the first lattice point in $x^c$.

## 6.5 Discretisation and Sampling Errors

In order to produce an accurate result, any numerical integration must proceed with a sufficiently small time step $h$. The difference between the correct result and the numerical result with a particular time step is known as the *discretisation error*. For deterministic equations, it is usually (see next paragraph) sufficient to reduce $h$ until the result converges to the desired accuracy. For stochastic equations this is not precisely correct, as a greater number of time steps will induce a different choice "noise" from the random number generators. This means that the two trajectories will be different members of the ensemble of possible trajectories, so they should not be expected to converge to each other. In order to verify the convergence of a stochastic integration, it is necessary to ensure that the same trajectory, or *path*, is examined with time steps of differing size. This can be done by averaging the noise contributions of a fine-grained path to generate the noise contributions to a coarse-grained path.

Many problems represent some localised interaction in a psuedo-infite space. In other words any radiation shed from the interaction should propagate away without ever being

reflected back into the region of interest. In such cases boundaries must be sufficiently far away from the important area of the field to have no effect on the result, and when periodic boundary conditions are used (these are in fact enforced with **xmds**) damping must be applied near the boundaries so as to absorb any such radiation. The lattice used in each transverse dimension must be sufficiently fine to be able to exhibit the details, yet not so fine as to cause memory problems - or cause the solution to take forever to compute. Further still, field details that are significantly smaller than those present at initialisation may evolve during the simulation, and the evolution of such detail will ultimately be affected by the finesse of the transverse lattice. Therefore simply reducing the time step alone is not a complete guarantee of accuracy. The results must be inspected to ensure that the transverse lattice was fine enough to contain the detail. If unsure then re-run the simulation with a finer lattice, as the evolution may be attempting to generate singularities.

Usually, meaningful results only come from stochastic integration by averaging moments of the fields over large numbers of paths. This leads to a second form of error for stochastic equations due to the standard error in the mean. This is known as the *sampling error*, and scales as the inverse square root of the number of paths taken.

# Part III

# User Manual

# 7

# Development and Program Structure

It only takes a few items of information to define a numerical simulation, yet the list of instructions necessary to get a computer to calculate the solution is vast in comparison. Going from the former to the latter is non-trivial. It can be done in a single step (as far as the user is concerned) with high level programming languages, but most computer programmers know from experience that the numerical solution can be calculated more efficiently by writing comparatively more code in a low level script, than less code in a high level script. This is because the programmer knows the purpose of the program, and can optimise it accordingly as the list of instructions is expanded. On the other hand a compiler can only perform optimisations that it knows are guaranteed to work regardless of the purpose of the program.

By restricting the scope of programs to those with a known range of purposes, one is able to write a compiler that generates more efficient output code. In other words the possible input code is constrained to a very small set of instructions which, even though they may be complex, have a well defined expansion into a simpler instruction set. Further, it is not necessary to define the expansion down to the level of assembly code – one only needs to define the expansion to a point where it no longer relies critically on the compiler's optimisation ability. This is how **xmds** works: transform from specialised input script to *efficient* C code, and then let an ordinary compiler do the rest.

In choosing the high level syntax we have attempted to employ some degree of foresight: the future is likely to see more interconnection of computer networks around the world, and information is already taking a standardised form for transfer. Until now this has primarily been HTML (Hyper Text Mark-up Language), but now there is increasing interest in an "extensible" mark-up language of which HTML is only a subset. This is known as XML (eXtensible Mark-up Language) [9, 10]. Also, a standard for data interchange has been developed at Caltech [2] which is a subset of XML: XSIL or eXtensible Scientific Interchange Language, and this has been chosen as the format for field data input/output.

Using XML for the input script has another significant advantage: the format for the simulation data takes on a tree-like structure which can be extended and made as complex as is necessary. This is the extensible property of XML. Appropriately named element tags are used to mark-up the essential data, and may be nested accordingly to group the information. The result is a logical human-readable script with data represented in a manner preserving both synthetic and sequential relationships.

**xmds** has undergone a few development iterations for the purpose of refinement and extension. There have been two particularly difficult aspects. The first was in defining an appropriate equation syntax. This syntax had to be in C code style since **xmds** merely transplants these equations into the output C code (which it then compiles), and yet it had to be capable of representing equations of the form shown in Equation (8.3). Further, it was desired that the C code form of such an equation must be the same regardless of the type of integration algorithm chosen. This difficulty was compounded by the desire to include split-operator algorithms, which process the linear and nonlinear terms entirely separately. The result that we have achieved here is quite elegant, though there are a few caveats with regard to writing the equations.

The second area of difficulty was enabling the user to define exactly what they wanted as output. The simplest solution would have been to save the raw field data to file as the simulation progressed, and the user left to process it afterward in their own preferred plotting and calculation package. However, with stochastic problems the user usually wishes to calculate the average of some property or *moment* of the field over many different *trajectories* or integration runs. Therefore it was necessary to enable to user to define such moments so that they may be calculated and averaged before the output data is written to file.

The source code for **xmds** is written in object oriented C, otherwise known as C++. The main routine is very simple, and its procedure is as shown on the left side in Figure 7.1. The right side is performed by the simulation data class.

A non-validating XML parser processes the input file and populates a node tree based on the Document Object Model [11]. This parser was written by the author, but there is little need to describe it in detail here. This node tree is then passed to a simulation class object which extracts its own relevant data, and in turn creates child objects to process the relevant sub-trees. The class structure of **xmds** is shown in Figure 7.2.

Once the input file has been processed, a "dry run" of the main sequence tree is called in order to evaluate the total number of samples requested for each output moment group. Finally the output code is generated. This is accomplished with a "tree walking" technique. To begin with the simulation object writes any include statements that are necessary. Then it writes any define statements particular to itself, and then calls each of its child objects to do the same. This process is continued down the tree until every object in the tree has written its define statements. Starting again at the simulation element, the process is repeated to write the global variables, the routine prototypes, and finally the routines themselves. This tree walking is performed by the Element class with the virtual functions, aptly titled, `writeDefines()`, `writeGlobals()`, `writePrototypes()`, and `writeRoutines()`. These routines are then overridden in the derived classes to write the code specific to each class.

As mentioned in the previous section, one of the primary areas of difficulty was developing

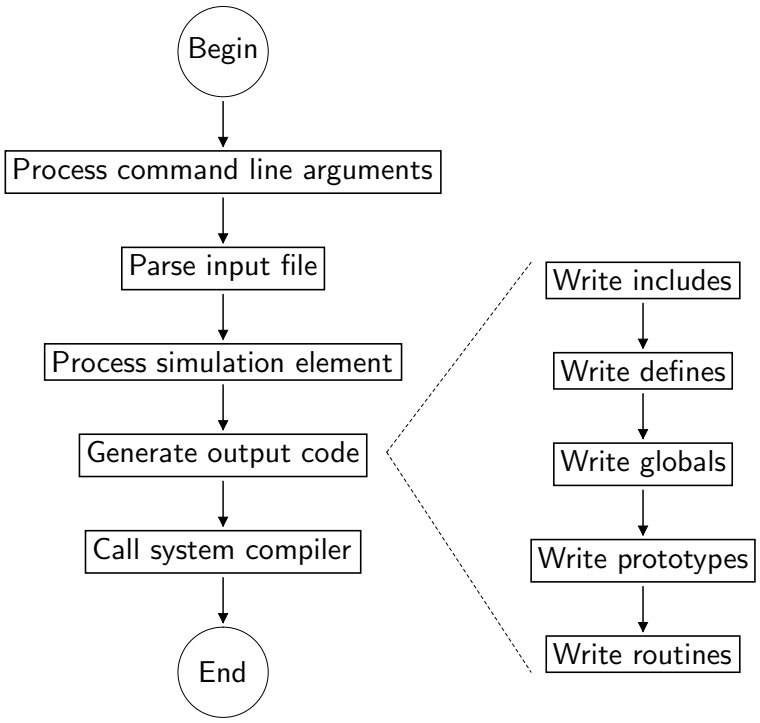



FIGURE 7.1: **xmds** main procedure

a C-code style syntax which would allow a large range of complex PDEs to be encoded, and in a manner that remains independent of the exact integrate algorithm chosen. This was implemented using a two step process. To begin with, the user is required to write equations with any differential operators replaced by an operator[component] representation. For example, the NLSE shown in Equation (7.1),

$$\frac{\partial \phi}{\partial z} = i\left[\frac{\partial^2 \phi}{\partial t^2} + |\phi|^2 \phi\right], \tag{7.1}$$

would be written in C-code style as:

```
dphi_dz =  L[phi] + i*~phi*phi*phi;
```

where the operator `L` is defined in Fourier space as being

```
L = -i*kt*kt/2;
```

The first step is to search the user's equations for such operator[component] expressions, and replace them with something else depending on the algorithm chosen. In the explicit picture they are replaced with a reference to an internally calculated vector (which is exactly the derivative calculated with the Fourier transform method), whereas in the interaction picture the field is evolved in Fourier space according to the operator[component] expressions found, and in the original code the text strings for these expressions are replaced with zero. The second step is to use `#define` macros to map the equation variables to the internal data arrays within the generated code. Thus for the example above, using an explicit picture method the output code in the derivative calculation routine becomes:

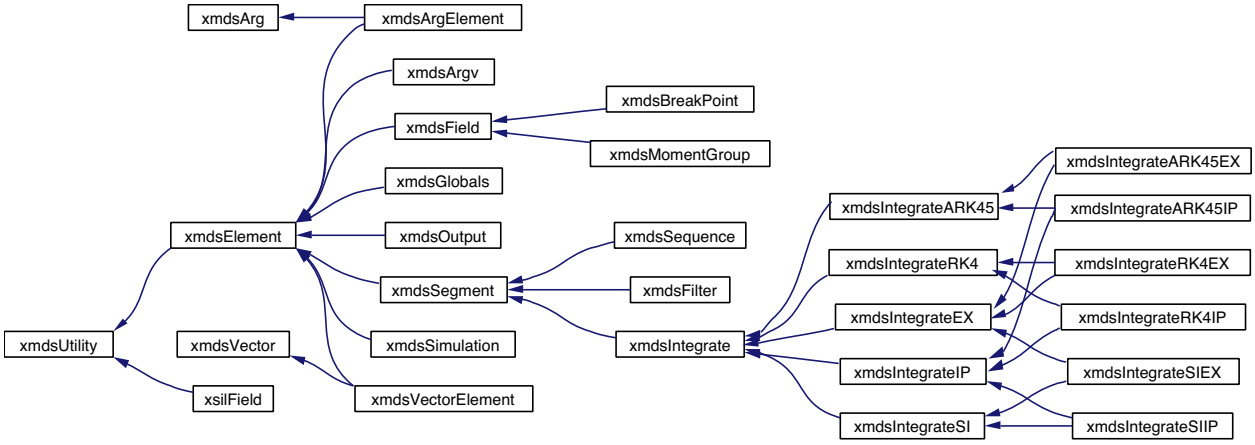



FIGURE 7.2: **xmds** class hierarchy

```
void _sg1_calculate_delta_a(
     const double& _step) {
complex dphi_dz;
_main_main_go_space(0);
_main_sg1_coterms_go_space(0);
unsigned long _main_main_pointer=0;
unsigned long _main_sg1_coterms_pointer=0;
double t = _main_xmin0;
for(unsigned long _i0=0; _i0<_main_lattice0; _i0++) {
//************** propagation code **************
dphi_dz =  _sg1_coterm_L_phi + i*~phi*phi*phi;
//*********************************************
     _main_main[_main_main_pointer + 0] = dphi_dz*_step;
     _main_main_pointer += _main_main_ncomponents;
     _main_sg1_coterms_pointer += _main_sg1_coterms_ncomponents;
     t += _main_dx0;
     }
}
```

with the following relevant define statements placed at the head of the file:

```
// field main defines
#define _main_ndims 1
#define _main_lattice0 100
#define _main_xmin0 -5.000000e+00
#define _main_dx0 ((5.000000e+00 - -5.000000e+00)/(double)100)
#define _main_dk0 (2*M_PI/(5.000000e+00 - -5.000000e+00))
#define dt _main_dx0
#define dkt _main_dk0
// vector main defines
#define _main_main_ncomponents 1
#define phi _main_main[_main_main_pointer + 0]
// vector sg1_coterms defines
```

```
#define _main_sg1_coterms_ncomponents 1
#define _sg1_coterm_L_phi _main_sg1_coterms[_main_sg1_coterms_pointer+0]
```

This does not make for compact code, but it was desired that the generated code remain readable by the user so that they may inspect it to see exactly what **xmds** is doing in regards to solving their problem. It also has the added advantage that, should a user wish to do something rare and obscure, they only need to write a short script to generate a base C-code listing which they may then alter to suit their problem.

# 8 Functionality

This chapter describes the full functionality of **xmds**, but in practice most of this detail is not required to integrate common types of ODEs and PDEs. One of the particularly useful features of **xmds** is that both the high-level description of the problem and the generated program can efficiently handle systems with widely varying complexity. Skip to the worked examples to see how very different kinds of equations can be easily integrated.

**xmds** is designed to integrate the following general PDE:

$$\frac{\partial}{\partial x^0}\mathbf{a}(\mathbf{x}) = \mathcal{N}\left(\mathbf{x}, \mathbf{a}(\mathbf{x}), \mathbf{p}(\mathbf{x}), \mathbf{b}(\mathbf{x}),\ \xi(\mathbf{x}\ , \int dx_j f(\mathbf{x}, \mathbf{a}(\mathbf{x}), \mathbf{p}(\mathbf{x}), \mathbf{b}(\mathbf{x}), \xi(\mathbf{x}))\right), \tag{8.1}$$

$$p^i(\mathbf{x}) = \mathcal{F}^{-1}\left[\Sigma_j \mathcal{L}^{ij}\left(x^0, \mathbf{k}_\perp\right)\mathcal{F}\left[a^j(\mathbf{x})\right]\right], \tag{8.2}$$

$$\frac{\partial}{\partial x^c}\mathbf{b}(\mathbf{x}) = \mathcal{H}\left(\mathbf{x}, \mathbf{a}(\mathbf{x}), \mathbf{b}(\mathbf{x})\right), \tag{8.3}$$

where the vector $\mathbf{a}$ represents an m-component real or complex valued field, though for some problems the field may only have one component. The vector $\mathbf{x}$ is the real-valued space in which $\mathbf{a}$ lies, which is the propagation dimension, $x^0$, plus the transverse dimensions, $x^{i\neq 0}$, if any. The number of transverse dimensions that the field may have is limited to 64 by **xmds**, but is more likely to be further limited by the complier and stack size used when the output code is compiled. The vector $\mathbf{p}$ may have any number of components (including zero), and may be derived through the action of a matrix of linear operators, $\mathcal{L}^{ij}$, on the main vector components. The action of these linear operators is calculated with the main vector in Fourier space, hence, as explained in Section 6.4.1, transverse partial derivatives reduce simply to multiplication by the transverse Fourier space dimensions. The vector $\mathbf{b}$ is optional, and is an n-component real or complex valued field which propagates along the transverse dimension $x^c$. For stochastic problems $\xi$ is a vector of independent real Gaussian noises. The total derivatives for vectors $\mathbf{a}$ and $\mathbf{b}$ are expressed as the general functionals $\mathcal{N}$ and $\mathcal{H}$ respectively. Note that this form allows for nonlinear partial derivatives that may

have spatial dependence, for example $\frac{1}{r}\left(\frac{\partial}{\partial r}\right)^2$.

Equation (8.3) is "Schrödinger-like" in that it is first order in the propagation dimension. As an example, if the field was a vector field representing three dimensional fluid flow then there would be three components to the field, which itself would exist in four dimensional space, with perhaps the propagation dimension being time and the three transverse dimensions being space.

**xmds** can only handle problems whose only *non-local* terms are integrals over some or all of the transverse dimensions. For local equations, the evolution of the field at any point in $\mathbf{x}$ is only ever a function (including derivatives) of the field components at that exact point in space—not anywhere else. This is clearly restrictive of the range of physical problems that may be modelled. However, many non-local problems are only so because of variables that propagate in a dimension which is transverse to the main propagation dimension, and thus are easily modelled by including a secondary field which is propagated orthogonally along this transverse dimension. This is the purpose of the $\mathbf{b}$ field which is the secondary or "cross" field propagating in the transverse dimension $x^c$, where $c \neq 0$.

In order to maximise the efficiency of the generated code while being able to handle a broad range of problems, **xmds** utilises a range of algorithms. These were covered in Section 6.4. Re-capping, the algorithms fell into two main categories:

1. **Explicit picture.** As detailed in Section 6.4.2 the transverse derivatives of the field are calculated using the Fourier Transform method of Section 6.4.1. In all cases this requires extra memory and computational expense, but it enables any equations of the general from of Equation (8.3) to be solved.

2. **Interaction picture.** Here the field components are also evolved in Fourier space using the interaction picture technique, as detailed in Section 6.4.8. The form of PDE that may be solved using this technique is shown in Equation (8.5).

$$\frac{\partial}{\partial x^0}a^i(x^0,\mathbf{x}_\perp) = \mathcal{F}^{-1}\left[\mathcal{L}^i(x^0,\mathbf{k}_\perp)\mathcal{F}[a^i(\mathbf{x})]\right] + \mathcal{N}^i\left(\mathbf{x},\mathbf{a}(\mathbf{x}),\mathbf{b}(\mathbf{x}),\xi(\mathbf{x})\right), \tag{8.4}$$

$$\frac{\partial}{\partial x^c}\mathbf{b}(\mathbf{x}) = \mathcal{H}\left(\mathbf{x},\mathbf{a}(\mathbf{x}),\mathbf{b}(\mathbf{x})\right). \tag{8.5}$$

   This form is significantly more restrictive than that of Equation (8.3). In particular, the linear operator matrix must be diagonal, and may not have coefficients with spatial dependence.

Within both of these pictures, either the semi-Implicit or Runge-Kutta algorithms may be employed. For more detail refer Section 6.4. This gives rise to the matrix of algorithms listed in Table 8.1.

There are two main things that **xmds** can do to a field: forward evolve (integrate/ propagate) it according to the set of PDEs, and reshape (filter) it according to a set of functions. These two actions may serve as the building blocks for an elaborate sequence of operations to be performed on the field. This is done by defining a sequence of *segments*; a segment being either an integrate step, a filter step, or else a sub-loop of further segments.

The key to **xmds**, as explained in the introduction, is that it is a *code generator*. **xmds** requires the user to write their particular PDEs (or ODEs) as a few lines of C code, which are

| | Explicit picture | Interaction picture |
|---|---|---|
| Semi-Implicit | SIEX | SIIP |
| 4th Order Runge-Kutta | RK4EX | RK4IP |
| 4th/5th Order adaptive Runge-Kutta | ARK45EX | ARK45IP |
| 9th Order Runge-Kutta | RK9EX | RK9IP |
| 8th/9th Order adaptive Runge-Kutta | ARK89EX | ARK89IP |

Table 8.1: The Algorithm Matrix

transplanted from the input script to the relevant points in the output code. This technique proves itself well for efficient code that is extremely flexible. A potential user should not be daunted at the prospect of having to learn basic C syntax – the flexibility gained by this approach well justifies the effort.

In terms of output results, it is usually desired to sample the field at various points throughout the sequence of operations and thus generate a sequence of samples. Further, when sampling, often the raw complex value of the field at the point in question is not relevant, it is a more general *moment* of field components and dimensions that is desired. The examples provided in Chapters 9 and 10 will help explain. Also, the evolution of the field is usually solved on a lattice much finer than is necessary to make a good plot of the output. Thus to save on memory and also size of output file it is better to sample on some reduced lattice rather than at every point in the main field lattice. It may also be desirable, when sampling for a particular group of moments, to transform the field to Fourier space in one, some or all of the transverse dimensions. At this point it is also possible to collapse one or more of the transverse dimensions by requesting either to sample a cross-section or else to integrate over the dimension in question with a particular kernel function. Once all field propagations are finished the sampled output can be post-processed to transform the propagation dimension into Fourier space, and again place the remaining transverse dimensions into any partial Fourier space, which may be the same or different to the one in which the field was sampled in. The original sampled moments may be used here or moments of these moments may be specified. Finally, if the evolution of the field involves stochastic terms, it will usually be desired to perform this sequence of operations a number of times (re-initialising the field each time) so as to determine averages and standard errors in those averages for the output moments over multiple trajectories, or *paths*.

Diagrammatically, the functionality of **xmds** is shown in Figure 8.1. Usually, using **xmds** is much simpler than this—look at the worked examples!

## 8.1 Installing and Running xmds

This software package is designed to install onto Unix and Linux operating systems (including the Cygwin environment on Windows), though an experienced programmer might easily port it to another platform. **xmds** requires a very small amount of disk space, and the RAM required is dependent algorithm, but is usually little more than the size needed to store a certain number of copies of the fields to be integrated. A C++ compiler is required as are also

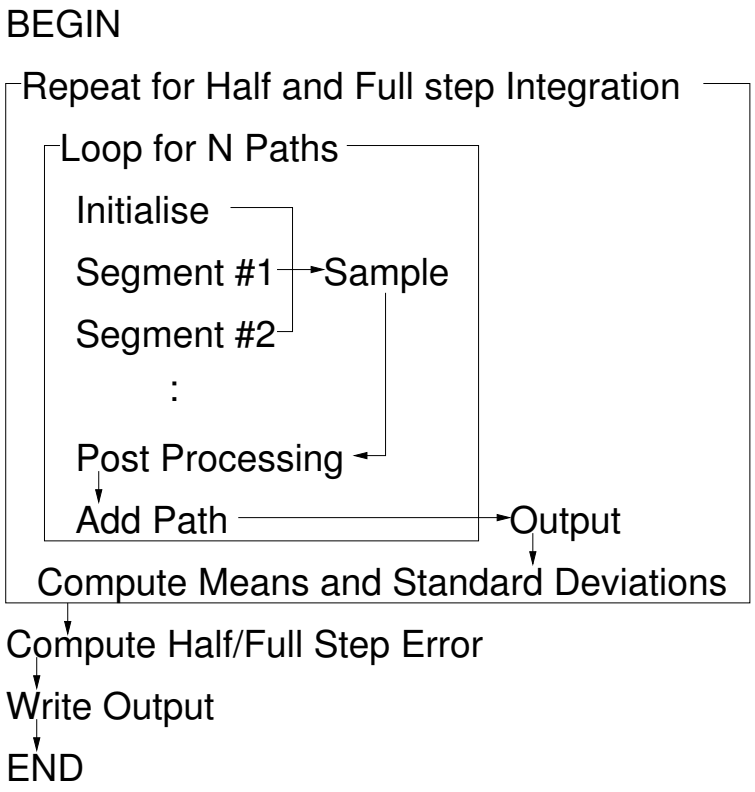



FIGURE 8.1: **xmds**—a functional diagram

the FFTW (Fastest Fourier Transform in the West) libraries for C. Further, for stochastic problems **xmds** can produce output code which uses MPI routines to parallelise the problem for running on multiple CPUs or computer clusters. An MPI compiler is required to take advantage of this. XMDS can also use an OpenMP threads to use multiple processors on single jobs if an OpenMP-enabled compiler is available.

### 8.1.1 Installation

Installation instructions and files are available at: http://www.xmds.org/downloads

First, if parallel processing is desired, a working MPI system and/or an OpenMP-enabled compiler should be installed.

Second, the FFTW libraries must be installed if they are not already present. These libraries may be found at: http://www.fftw.org/. Version 2.1.x (and, optionally 3.x) of the fftw library is required for proper execution of **xmds**. **xmds** expects that the `fftw.h` header file is in one of the `/usr/include` or the `/usr/local/include` directories, with the libraries in a sibling `/usr/lib` or `/usr/local/lib` directory. These are the standard locations. If the fftw installation is non-standard then you will need to use the`--with-fftw-path` option for the `configure` program, pointing to the parent directory containing the fftw `/include` and `/lib` sub-directories. If parallel processing is desired, FFTW must be built with MPI or OpenMP options enabled.

The main method of installation is from a "tarball". Download `xmds-1.6.5.tar.gz` and

then (as root) [1]:

```
tar -xzvf xmds-1.6.5.tar.gz
cd xmds-1.6.5
./configure
make
make install
```

or as a user (to be installed in the `bin` directory of your home directory):

```
tar -xvzf xmds-1.6.5.tar.gz
cd xmds-1.6.5
./configure --with-user
make
make install
```

For help on the various configuration options, one can run the command

```
./configure --help
```

which will show a (very long) list of options which can be changed to customise the way **xmds** is installed and the options that it can use to build simulations. The different variables that one can change are:

- `XMDS_CC`: the C++ compiler used by **xmds** to compile simulations
- `XMDS_CFLAGS`: the C++ compiler flags used to build simulations
- `XMDS_LIBS`: the libraries used to build simulations
- `XMDS_INCLUDES`: the include flags used for building simulations
- `FFTW_LIBS`: the libraries specific to `fftw`
- `FFTW_MPI_LIBS`: the libraries specific to `fftw` but for the MPI compiler
- `FFTW3_LIBS`: the libraries specific to version 3 of `fftw`
- `MPICC`: the MPI C++ compiler
- `MPICCFLAGS`: the compiler flags to use with `mpicc`
- `MPILIBS`: the library flags to pass to `mpicc`
- `USER_LIB`: the location of the file **xmds** library functions, only necessary if **xmds** is compiled for user use (e.g. `-L/home/cochrane/bin`)
- `USER_INCLUDE`: the location of the **xmds** header files, only necessary if **xmds** is compiled for user use (e.g. `-I/home/cochrane/bin`)

[1] **Note:** when `fftw` is installed in `/usr/local` then `--with-fftw-path` must be specified so that the configure script can find the libraries and headers.

One can also set various options on the command line as arguments to the configure script. The options available are:

- `--with-user`: to install **xmds** into the user's `bin` directory within their home directory. This will also point the `USER_LIB` and `USER_INCLUDE` variables to the correct places so that **xmds** will find its required headers and libraries so that it can build simulations.
- `--enable-mpi`: to check for MPI in the configuration and to enable the use of MPI for building simulations.
- `--enable-threads`: to check for the ability of the fftw libraries to use threads, and enable their use in simulations.
- `--enable-fftw3`: to check for version 3 of fftw and enable use of it for building simulations.
- `--with-fftw-path`: the path to the `fftw` installation. **Note:** when `fftw` is installed in `/usr/local` then `--with-fftw-path` must be specified so that the configure script can find the libraries and headers.
- `--with-fftw3-path`: the path to the `fftw3` installation (if different to the path for fftw2).
- `--with-mpi-libs`: extra libraries needed when checking for MPI.
- `--with-mpi-path`: set the path to the prefix of your MPI distribution.
- `--with-mpi-compiler`: set the mpi C++ compiler.

### 8.1.2 Usage

For usage, at the command prompt type:

```
% xmds
```

and the output should be:

```
This is xmds version 1.6.5,

        using C compiler 'gcc'
        (and C compiler 'mpicc' for parallel work)


Usage: xmds [-v] [-c] infile
        infile: required,  the input file
             v: optional,  verbose mode
             c: optional,  turns off automatic compilation of simulation
```

or something similar. The "verbose" mode can be quite useful as it will repeat back to the user exactly how it is interpreting the input file. Switching off automatic compilation can be handy when one is testing a script and doesn't want to wait for the script to compile, especially if the resultant binary isn't wanted anyway. This option can also be used by those people who want to use **xmds** to produce a skeleton C++ code with which they want to change directly themselves to perform something more complex than is yet possible with **xmds**.

The first thing **xmds** does is run the input file through its own XML parser. If the input file contains bad XML syntax then these will be the first errors to be picked upon. **xmds** then processes the input file, and writes the code dedicated to solving the particular simulation, and compiles it.

```
% xmds nlse.xmds
compiling ...
     gcc    -pthread -O3 -ffast-math -funroll-all-loops
     -fomit-frame-pointer -o nlse nlse.cc  -I/home/cochrane/bin
     -lstdc++ -lm -lxmds -L/home/cochrane/bin -lfftw_threads -lfftw
'nlse' ready to execute
```

All that remains is to execute the compiled program:

```
% nlse
```

Refer to the `kubo.xmds` example in Section 10.2 for execution of a parallel problem. Or see Chapter 4.

### 8.1.3 Preferences

As of `xmds-1.3-1` it is possible for people to specify the simulation build options in a preferences file. Previously, if one wanted to change how the simulations were built, one had to recompile **xmds** with the relevant compile flags etc. Now, all one has to do is specify the compilation flags in a preferences file. **xmds** looks for a file called `xmds.prefs` in `$HOME/.xmds/` and (if not found there) in the directory local to the xmds script you are trying to compile. The format of the preferences file is:

```
<compile flag name> = <compile flag value>
```

For example:

```
XMDS_CC = gcc
```

Comments can be added by using a hash character (`#`). Any thing after (and including) the hash are ignored. As another example of a preferences file, here is an example `xmds.prefs`:

```
# xmds preferences file
XMDS_CC=gcc  # this, here, is a comment
XMDS_CFLAGS = -pthread -O3 -ffast-math -funroll-all-loops
XMDS_LIBS=-lstdc++ -lm -lxmds -L/home/cochrane/bin
XMDS_INCLUDES = -I/home/cochrane/bin
THREADLIBS = -lfftw_threads
```

The use of preferences can be switched on and off with the use of the <use_prefs> tag (which should be located after the <simulation> element, near where <error_check> and friends live). This is a boolean option, so to switch preferences off one should use no. The default is yes, but if you don't have any preferences, or you haven't specified them all, then the default values (that were decided when **xmds** was built) are used for anything not specified.

## 8.2 Syntax summary

The xml elements used can be divided into two main categories: those that may contain code and/or other elements, and those that merely contain variable assignments. The former can perhaps be called "structural" elements, and the latter perhaps "assignment" elements. These are summarised in Tables 8.2 and 8.3 respectively.

| Element name | Used in | Req. | May contain |
|---|---|---|---|
| <simulation> | top level | yes | <name>, <prop_dim>, <error_check>, <stochastic>, <paths>, <noises>, <argv> <globals>, <field>, <sequence>, <output> <binary_output>, <use_double>, <use_wisdom> <benchmark>, <use_prefs>, <argv>, <threads> <fftw_version> |
| <globals> | <simulation> | no | C code |
| <argv> | <simulation> | no | <arg> |
| <arg> | <argv> | yes | <name>, <type>, <default_value> |
| <field> | <simulation> | yes | <name>, <dimensions>, <lattice> <domains>, <samples>, <vector> |
| <vector> | <field> | yes | <name>, <type>, <components>, <vectors> <filename>, <fourier_space>, and C code |
| <sequence> | <simulation> | yes | <integrate>, <filter>, <sequence> |
| | <sequence> | no | <cycles>, <integrate> <filter>, <sequence> |
| <integrate> | <sequence> | no | <algorithm>, <interval>, <lattice>, <samples>, <k_operators>, <vectors>, <cross_propagation>, <iterations>, <moment_group>, <tolerance>, <max_iterations>, <min_time_step>, <cutoff>, <smallmemory>, <no_noise>, <halt_non_finite>, and C code |
| <k_operators> | <integrate> | no | <constant>, <operator_names>, and C Code |
| <cross_propagation> | <integrate> | no | <prop_dim>, <vectors>, and C Code |
| <filter> | <sequence> | no | <fourier_space>, <vectors>, <functions>, <moment_group>, and C Code |
| <output> | <simulation> | yes | <filename>, <group> |

| | | | |
|---|---|---|---|
| <group> | <output> | yes | <sampling>, <post_propagation> |
| <sampling> | <group> | yes | <lattice>, <fourier_space>, <vectors>, <moments>, <type>, and C code |
| <post_propagation> | <group> | no | <fourier_space>, <moments>, and C Code |

Table 8.2: The structural elements

| Element name | Used in | Req. | May contain |
|---|---|---|---|
| <name> | <simulation> | no | string: defaults to filename-extn |
| | <field> | no | string: defaults to "main" |
| | <vector> | yes | string |
| | <arg> | yes | string |
| <prop_dim> | <simulation> <cross_propagation> | yes | string |
| <error_check> | <simulation> | no | yes/no: defaults to yes |
| <stochastic> | <simulation> | no | yes/no: defaults to no |
| <use_mpi> | <simulation> | no | yes/no: defs. to no |
| <MPI_Method> | <simulation> | no | Scheduling/Uniform: defs. to Scheduling |
| <paths> | <simulation> | – | integer: reqd. if stochastic |
| <seed> | <simulation> | – | integer: 2 reqd. if stochastic |
| <noises> | <simulation> | – | integer: reqd. if stochastic |
| <dimensions> | <field> | no | array of strings |
| <lattice> | <field> <sampling> | – | array of integers, same no. as trans. dims |
| | <integrate> | yes | integer |
| <domains> | <field> | – | array of bracketed pairs of floats |
| <components> | <vector> | yes | array of strings |
| <type> | <vector> <sampling> | no | <complex> or <double> |
| | <arg> | yes | string |
| <vectors> | <vector> <k_operators> <sampling> | no | array of strings |
| | <integrate> <filter> <cross_propagation> | yes | array of strings |
| <moment_group> | <integrate> <filter> | no | array of strings |
| <functions> | <integrate> <filter> | no | array of strings |
| <constant> | <k_operators> | no | yes/no: defaults to no |

| | | | |
|---|---|---|---|
| <operator_names> | <k_operators> | yes | array of strings |
| <filename> | <vector> | no | string |
| | <output> | no | string: defaults to <name>.xsil |
| <fourier_space> | <vector>, <filter> | no | array of yes/no |
| | <sampling> | yes | array of yes/no |
| | <post_propagation> | yes | array of yes/no: same no. as non-collapsed trans. dims + 1 |
| <samples> | <field> <integrate> | yes | integer: as many as are <group> |
| <cycles> | <sequence> | yes | integer (not in main sequence) |
| <algorithm> | <integrate> | no | string: defaults to RK4EX (or SIEX for stochastic problems) |
| <interval> | <integrate> | yes | string |
| <no_noise> | <integrate> | no | yes/no, defaults to no |
| <iterations> | <integrate> | no | (SIIP: integer, defaults to 3) |
| <tolerance> | <integrate> | yes | (ARK45: string) |
| <max_iterations> | <integrate> | no | (ARK45: integer, defaults to infinity) |
| <min_time_step> | <integrate> | no | (ARRK45/89: double, defaults to 1e-13) |
| <cutoff> | <integrate> | no | (ARK45: string, defaults to 1e-3) |
| <smallmemory> | <integrate> | no | (ARK45IP: yes/no, defaults to no) |
| <halt_non_finite> | <integrate> | no | yes/no: defaults to no |
| <moments> | <sampling> <post_propagation> | yes | array of strings |
| <default_value> | <arg> | yes | depends upon <type> declaration |
| <benchmark> | <simulation> | no | yes/no: defaults to no |
| <use_wisdom> | <simulation> | no | yes/no: defaults to no |
| <binary_output> | <simulation> | no | yes/no: defaults to no (**deprecated**) |
| <use_double> | <simulation> | no | yes/no: defaults to no (**deprecated**) |
| <use_prefs> | <simulation> | no | yes/no: defaults to yes |

Table 8.3: The assignment elements

## 8.3 C-coding within elements

**xmds** is all about transplanting equations into the necessary multi-dimensional loop structure to solve them, which has the direct consequence that it is necessary for your equations to be written in standard C syntax. To begin with, variable names may not start with a number, and may not be a reserved keyword such as *void, int, long, double, complex, for, while, if, switch, return, etc....* This applies to the variable names listed in the <dimensions>, <components>, <operator_names>, and <moments> assignments. It also applies to user defined variables where C-code is allowed.

| Function name | Math | Argument(s) | Result |
|---|---|---|---|
| `abs(n)` | $\|n\|$ | integer | integer |
| `fabs(x)` | $\|x\|$ | real | real |
| `exp(x)` | $e^x$ | real | real |
| `log(x)` | $ln(x)$ | real | real |
| `sqrt(x)` | $\sqrt{x}$ | real | real |
| `pow(x,y)` | $x^y$ | real,real | real |
| `sin(x)` | $sin(x)$ | real | real |
| `asin(x)` | $sin^{-1}(x)$ | real | real |
| `cos(x)` | $cos(x)$ | real | real |
| `acos(x)` | $cos^{-1}(x)$ | real | real |
| `tan(x)` | $tan(x)$ | real | real |
| `atan(x)` | $tan^{-1}(x)$ | real | real |
| `cot(x)` | $cotan(x)$ | real | real |
| `complex(x,y)` | $x + iy$ | real,real | complex |
| `rcomplex(x,y)` | $x + iy$ | real,real | complex |
| `pcomplex(x,y)` | $xe^{iy}$ | real,real | complex |
| `conj(z)` or `z` | $z^*$ | complex | complex |
| `c_exp(z)` | $e^z$ | complex | complex |
| `c_log(z)` | $ln(z)$ | complex | complex |
| `c_sqrt(z)` | $\sqrt{z}$ | complex | complex |
| `real(z)` | $real\{z\}$ | complex | real |
| `imag(z)` | $imag\{z\}$ | complex | real |
| `mod(z)` | $\|z\|$ | complex | real |
| `arg(z)` | $imag\{ln(z)\}$ | complex | real |
| `mod2(z)` | $\|z\|^2$ | complex | real |

Table 8.4: Commonly used functions

Globals should be declared as:

```
const long int1 = 7;
const double real1 = 23.45;
const complex comp1 = complex(0.4,0.2);

double r = sqrt(x*x + y*y);
complex z = pcomplex(r,M_PI/2);
```

As can be seen there are three main variable types available: integers, reals, and complex – best declared as `long`, `double`, and `complex` respectively. The range of mathematical functions available is the same as that for C-programming, as well as some complex functions defined in the `xmdscomplex.h` header file in the `/source` directory. The ones most likely to be of use are summarised in Table 8.4. And remember, avoid using transcendental functions in the main integration equations if speed is important.

| | Example for:<br><prop_dim>z</prop_dim><br><noises>2</noises><br><dimensions>x y</dimensions> |
|---|---|
| Variable | Available in |
| z | <integrate>, <k_operators>, <filter>, <sampling> |
| z or kz | <post_propagation> |
| dz | <integrate>, <k_operators> |
| x or kx<br>y or ky<br>dx or dkx<br>dy or dky | <vector>, <integrate>, <k_operators>, <sampling>, and <post_propagation> where dimension hasn't been collapsed |
| n_1, n_2 | <vector>, <integrate>, <filter> |

Table 8.5: Automatically declared variables

It is also possible to create piece-wise functions by using `if` statements, which must be done in the usual C syntax. Here are two alternate syntaxes that will turn on functions `damping1` and `damping2` after `t=1.0` and `t=2.0` respectively:

```
double damping1;

if(t>1.0)
  damping1 = 1.0;
else
  damping1 = 0;

const double damping2 = t>2.0? 1 : 0;
```

**xmds** automatically declares a number of variables, depending on the element, for the equations to reference. These are summarised in Table 8.5. If a particular dimension is in Fourier space (ie. if the corresponding `<fourier_space>` assignment was "`yes`") then it is referenced by its name as specified in the `<field>` element but prefixed by a "`k`".

Finally, be wary of unintentional integer divisions, for example `const double c = 1/2;`, as the compiler will perform an integer division on the `1/2` before converting to a double, resulting in `c=0.0`! It ought to be written as `const double c = 1.0/2;` if `c=0.5` was the intended result.

# 9 Worked example: nlse.xmds

XML is very much like HTML, so readers with any experience in writing HTML will find XML fairly straight forward. However, even if a user has not had any experience in HTML they will pick up the XML with the help of some example simulation files. A comprehensive guide to general XML syntax is available in [9]. For now, a good introduction will be to step through the "`nlse.xmds`" example script, then we can work through some more advanced examples to elaborate on some of the options available. These examples scripts can be found in the `/examples` directory, along with many others. They may also be downloaded from the web site http://www.xmds.org/examples.html.

```
<?xml version="1.0"?>
<!--Example simulation: Non-Linear Schroedinger Equation-->

<simulation>

  <name>nlse</name>
  <prop_dim>z</prop_dim>
  <error_check>yes</error_check>
  <stochastic>no</stochastic>

  <globals>
    <![CDATA[
      const double energy = 4;
      const double vel = 0.0;
      const double hwhm = 1.0;
    ]]>
  </globals>

  <field>
```

```
    <name>main</name>
    <dimensions>  t   </dimensions>
    <lattice>    100  </lattice>
    <domains>  (-5,5) </domains>

    <samples>1</samples>
    <vector>
      <name>main</name>
      <type>complex</type>
      <components>phi</components>
      <fourier_space>no</fourier_space>
      <![CDATA[
        const double w0 = hwhm*sqrt(2/log(2));
        const double amp  = sqrt(energy/w0/sqrt(M_PI/2));

        phi = pcomplex(amp*exp(-t*t/w0/w0),vel*t);
      ]]>
    </vector>
    <vector>
      <name>vc1</name>
      <type>double</type>
      <components>damping</components>
      <fourier_space>no</fourier_space>
      <![CDATA[
        damping=1.0*(1-exp(-pow(t*t/4/4,10)));
      ]]>
    </vector>

  </field>

  <sequence>
    <integrate>

      <algorithm>RK4IP</algorithm>
      <interval>20</interval>
      <lattice>1000</lattice>
      <samples>50</samples>
      <k_operators>
        <constant>yes</constant>
        <operator_names>L</operator_names>
        <![CDATA[
          L = rcomplex(0,-kt*kt/2);
      ]]>
      </k_operators>
      <vectors>main vc1</vectors>
      <![CDATA[
        dphi_dz =  L[phi] + i*~phi*phi*phi - phi*damping;
```

```
      ]]>
    </integrate>
  </sequence>

  <output>
    <filename>nlse.xsil</filename>
    <group>
      <sampling>
        <fourier_space> no </fourier_space>
        <lattice>       50 </lattice>
        <moments>pow_dens</moments>
        <![CDATA[
          pow_dens=~phi*phi;
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

The above XML file describes to **xmds** how to write a program that solves the Nonlinear Schrödinger Equation in one dimension; Equation (9.1).

$$\frac{\partial \phi}{\partial \xi} = i\left[\frac{1}{2}\frac{\partial^2 \phi}{\partial \tau^2} + i\Gamma(\tau)\phi + |\phi|^2\phi\right]. \tag{9.1}$$

Here $\phi$ is the single component complex field (`phi` in the input file), $\xi$ is the propagation dimension (`z`), $\tau$ is the local time coordinate (`t`), and $\Gamma$ is a damping field (`damping`) applied to absorb scattered radiation at the domain boundaries. Figure 9.1 shows what the output of this simulation should look like.

At the head of the file:

```
<?xml version="1.0"?>
<!--Example simulation: Non-Linear Schroedinger Equation-->
```

Here the first line defines the file as an XML file. **xmds** requires correct XML syntax, but otherwise does not need to validate the file with a DTD (Document Type Definition). The second line is simply a comment line. All comments begin with a "`<!--`" and end with a "`-->`". Any characters are allowed in-between save for the comment closing sequence "`-->`".

The remainder of the file is covered in detail in Sections 9.1–9.8, while Section 8.2 contains a tabulated summary of the syntax.

## 9.1 The simulation element

```
<simulation>

  <name>nlse</name>
```

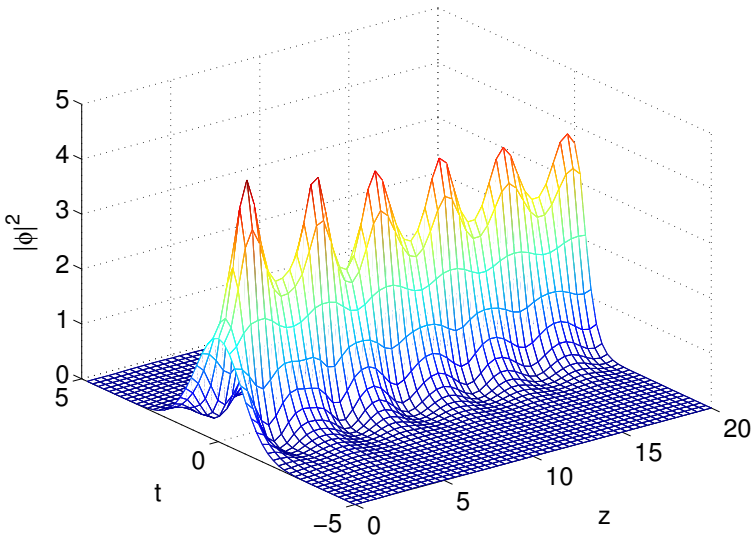



FIGURE 9.1: Results for nlse.xmds

```
  <prop_dim>z</prop_dim>
  <error_check>yes</error_check>
  <stochastic>no</stochastic>

  <!-- More xmds code -->

</simulation>
```

The root XML element of the input file must be a `<simulation>` element, which contains the high level description of the problem.

Within the `<simulation>` element **xmds** will first look for a simulation `<name>`. This information is optional, the default being the file name minus its last extension. It is used as the base for the output `.cc` file, and is the name of the executable that **xmds** will make.

The `<prop_dim>` assignment defines the name of the main propagation dimension. It is compulsory, as it is expected that whichever symbol is used here will also appear in the equations the user supplies.

The `<error_check>` assignment is optional, the default being "`yes`". This option tells **xmds** whether to write the code so as to repeat the simulation with half the step sizes in the integration segments, and subsequently compare the two sets of results. It is a good idea to leave this option on until confident that the errors are acceptable for the simulation in question. If set to "`yes`" the output executable will compare all output moments on a point by point basis between the full and half-step runs. It will then write out to screen the maximum value of the discrepancy, and the half-step results will be used for the final output.

The `<stochastic>` assignment is also optional, the default being "`yes`". It tells **xmds** whether the simulation uses Gaussian noise terms or not. If `<stochastic>` is set to "`yes`" then three further assignments, `<paths>`, `<seed>`, and `<noises>` become compulsory. The `kubo.xmds` and `fibre.xmds` examples in Chapter 10 cover stochastic simulations in more

detail.

The child elements within the <simulation> element that **xmds** then looks for are <globals>, <field>, <sequence>, and <output>.

## 9.2 The globals element

```
<globals>
  <![CDATA[
    const double energy = 4;
    const double vel = 0.0;
    const double hwhm = 1.0;
  ]]>
</globals>
```

This element is optional, and is used to define any numerical constants that are useful to have globally available to all sections of code. The <![CDATA[ ... ]]> container tells the XML parser to copy its contents without attempting to parse them. The contents:

```
const double energy = 4;
const double vel = 0.0;
const double hwhm = 1.0;
```

are copied directly into the output C code and therefore must have correct C syntax. See Section 8.3 for more on C syntax.

## 9.3 The field element

```
<field>
  <name>main</name>
  <dimensions>  t   </dimensions>
  <lattice>    100  </lattice>
  <domains>  (-5,5) </domains>

  <samples>1</samples>

  <!-- Possibly more xmds code -->

</field>
```

This element is compulsory as it is central to the problem definition: it specifies the geometry of the field. There are five assignments that **xmds** looks for in this element: <name>, <dimensions>, <lattice>, <domains>, and <samples>.

The <name> assignment provides a name for the field. It is not compulsory and will default to "main". It is envisaged that future versions **xmds** may allow sub fields within fields, but at present only one <field> element is permitted.

The <dimensions> assignment lists the names of the transverse field dimensions. From this point on the number of transverse dimensions is set to the number of names found in this assignment. If this assignment is empty or absent then the field is assumed to be without transverse dimensions (as in the kubo.xmds example), and **xmds** will not look for either

a <lattice> assignment or a <domains> assignment. The dimension names must be valid names as far as the C language is concerned–see Section 8.3.

The <lattice> assignment is compulsory if there were one or more transverse dimension names found in the <dimensions> assignment, and **xmds** will look for the same number of positive integers here as there were transverse dimensions. Each of these integers defines the number of points or steps to use in the corresponding transverse dimension, and so **xmds** expects to find integers with a value of 2 or more in each case.

The <domains> assignment is similar, except that it defines the domain range for each dimension. This is done by entering a bracketed pair of real numbers for each dimension, and **xmds** expects the range $b-a$ for any pair (a,b) to be greater than $10^{-100}$. Each domain range is divided according to Equation (9.2), in which $n$ is the number of lattice points. Note that the points stop one spacing short of the upper value—this is due to the forced periodic boundary conditions.

$$x_j = \frac{(n-j+1)a + (j-1)b}{n}, \quad j = 1, \ldots, n. \tag{9.2}$$

Finally the <samples> assignment tells **xmds** which moment groups (Section 9.8) to sample immediately after all the field's vectors have been initialised. A sequence of "0" or "1" entries is expected, as many as there are moment groups defined in the output (**xmds** checks this once all moment groups have been processed). The correspondence is sequential. A "1" means sample the moment group in question, a "0" means do not.

Next **xmds** will look for the field's vectors.

## 9.4 The vector element

```
<vector>
  <name>main</name>
  <type>complex</type>
  <components>phi</components>
  <fourier_space>no</fourier_space>
  <![CDATA[
    const double w0 = hwhm*sqrt(2/log(2));
    const double amp  = sqrt(energy/w0/sqrt(M_PI/2));

    phi = pcomplex(amp*exp(-t*t/w0/w0),vel*t);
  ]]>
</vector>
```

The <name> assignment is compulsory here, as it is quite common to employ more than one vector. The field must have at least one vector called "main", as it is this vector that **xmds** forward evolves in the main propagation dimension.

The <type> assignment is optional. It will default to "complex" unless specifically stated as "double", which is the standard data type in C for floating point variables. Note that vectors of type "double" cannot be Fourier transformed without some arbitrary definition of how this is done. Thus **xmds** requires that "double" vectors always remain in the Fourier

space in which they were initialised. A request to access them in some other Fourier space will result in an error.

The <components> assignment is compulsory. It identifies by name the components of the vector. Here there is only one component, but more may be defined simply by separating them with spaces, for example <components>phi theta</components>. The component names must be valid as far as the C language is concerned – see Section 8.3.

Finally the initialisation of the vector must be defined. This can be done in two ways. Either by means of a <filename> assignment, or by supplying C code. Here we are doing the latter. In the absence of <filename> assignment **xmds** looks for a <fourier_space> assignment followed by the initialisation code.

The <fourier_space> assignment must contain as many "yes" or "no" entries as there are transverse dimensions in the field, which in this case is only one. A "yes" entry means the corresponding dimension is in Fourier space when initialised, and vice-versa.

When initialising from code, the code is included as the content of the <vector> element. There must be an equation for each component, and these may include functions of <global> variables and of the transverse dimensions. Again see Section 8.3 for more information. Two points to remember: firstly if the field components are complex variables they may be initialised using the complex functions in Table 8.4, and secondly every field component should be explicitly initialised – even if it is only being initialised to zero.

Finally, when initialising from file, **xmds** expects that the file is simply a sequence of numbers in ASCII format, with as many numbers as the product of lattice points and vector components. In usual C convention, the component index varies most rapidly, then the right most transverse dimension lattice and so on. The left most transverse dimension lattice dimension varies most slowly. If the vector is complex then each data point is read in as a sequential real and imaginary pair. All vectors initialised from file are assumed to be entirely in normal space.

## 9.5 The sequence element

```
<sequence>

  <integrate>

    <!-- More xmds code -->

  </integrate>

  <!-- More xmds code -->

</sequence>
```

Next comes the <sequence> element, which can be used in two different contexts. The first context (as used here) is the "top level" <sequence> element, and is simply for defining the sequence of segments that happen to the field after initialisation. In a stochastic simulation this sequence is repeated for each integration path, but in this non-stochastic example the sequence is only stepped through once. In this context **xmds** expects to find any number

of <integrate>, <filter>, or <sequence> elements – which leads us to the second context.

In the second context <sequence> elements may be used as child elements within parent <sequence> elements, in which case they represent a sub-loop to be repeated one or more times. The number of repetitions is defined by a simple <cycles> assignment within, which is expected to be the first element. In the absence of a <cycles> element the number of cycles defaults to one, but if it is present it must come first. Any number of <integrate>, <filter>, or <sequence> elements may then follow. Obviously the ordering of segments within any sequence element is important.

## 9.6 The integrate element

```
<integrate>

  <algorithm>RK4IP</algorithm>
  <interval>20</interval>
  <lattice>1000</lattice>
  <samples>50</samples>
  <k_operators>
    <constant>yes</constant>
    <operator_names>L</operator_names>
    <![CDATA[
          L = rcomplex(0,-kt*kt/2);
    ]]>
  </k_operators>
  <vectors>main vc1</vectors>
  <![CDATA[
    dphi_dz =  L[phi] + i*~phi*phi*phi - phi*damping;
  ]]>
</integrate>
```

Here lies the heart of what **xmds** is all about, and not surprisingly it forms the most complex element. The <algorithm> assignment is optional, and will default to SIEX for stochastic simulations and RK4EX for non-stochastic simulations. The assignments <interval>, <lattice>, and <samples>, are all compulsory, and respectively represent the integration range, total number of steps, and number of samples for each output moment group to take within these steps. Thus each integer in the <samples> assignment must be either zero or else a factor of <lattice>, and if not **xmds** will exit with an error message to that effect.

The rest of the <integrate> element consists of "writing down" the form of Equation (8.3) in a high level format. Here the algorithm is RK4IP, which, as explained earlier, is an interaction picture algorithm. The dispersion term in Equation (9.1) is represented here by the linear operator "L" acting on field component "phi". The presence of the term "L[phi]" in the equations causes the field to be evolved in Fourier space in accordance with the algorithm listed in Section 6.4.10. The $\mathcal{N}$ operators are everything left over once terms such as this are removed.

The <k_operators> element contains just what its name implies: k-space operators. Within this element a <constant> declaration is used to indicate whether or not these operators depend on the propagation dimension. This declaration is optional with the default being "no", but this will produce slower code. If the operators do not depend on the propagation dimension then you will get faster code is you enter <constant>yes</constant>. An <operator_names> assignment is compulsory. One or more operator names may be listed here, and they all must be explicitly defined in the code. Refer to Equation (6.33) for how to write the $\mathcal{L}^{ij}$ operators as functions of the Fourier space dimensions. The operator names must be valid names as far as the C language is concerned—see Section 8.3.

If there are no non-zero linear operators $\mathcal{L}^{ij}$, then the entire <k_operators> element may be omitted.

**Extreme DANGER!!!!!!!**

Because of the way **xmds** works internally, it is possible when using an interaction picture routine to write the main equations in such as way as that everything works fine, but the results will be incorrect! The problem is that **xmds** does not closely inspect the main equations. What it does is it looks for operator[component] combinations, registers their presence, and the replaces their text with something else. In the explicit picture they are replaced with a reference to an internally calculated vector ($\mathbf{p} = \mathcal{F}^{-1}\left[\mathcal{L}\left(x^0, \mathbf{k}_\perp\right) \cdot \mathcal{F}\left[\mathbf{a}\right]\right]$), but in the interaction picture they are replaced with "0" and the component $a^i$ being acted on by the operator $\mathcal{L}$ is evolved in Fourier space using $a^i = e^{\frac{1}{2}h\mathcal{L}^i\left(x^0,\mathbf{k}_\perp\right)}a^i$. This means that if you write

```
<![CDATA[
  dtheta_dz =  L1[theta] + i*theta*theta*theta - theta*damping;
  dphi_dz   =  L2[theta] + i*~phi *phi  *phi   - phi  *damping; // no !!
]]>
```

the component `theta` will be evolved in Fourier space using the sum of the operators `L1` and `L2`, and the evolution of `phi` will not include `L2[theta]`. Further, writing something like `3*L1[theta]` does not have the intended effect - `theta` will still only be evolved with `L1`, not `3*L1`. We realise this is a potential source of error, but it enables the equation syntax to remain uniform for all algorithms. A future version of **xmds** will check the users equations more thoroughly. When using explicit picture algorithms none of these problems exist. If no errors are reported then what you get will be what your equations asked for.

Sometimes, it is useful to calculate a function without having to calculate it separately for every transverse position. This can be done by including the code in a <functions> element.

```
<functions>
  <![CDATA[
        double f = sin(2.0*z);  // This is a function of the propagation dim
  ]]>
</functions>
```

Conversely, it is often desirable to reference a variable in the equations that is a function of the spatial coordinates, but is otherwise constant. The variable `damping` is exactly one of these. Rather than re-calculating it at every time step and lattice point, what we did here was to

define an additional field vector "vc1" in the <field> element that could store this variable, calculate it once when it is initialised, and then reference it in the main equations. Therefore, we must tell **xmds** which vectors we wish to access in the equations. This is done with the <vectors> assignment. The reason that we do not make *all* vectors available everywhere by default is one of efficiency. For example if a particular vector was initialised in x-space then it would have to be transformed to k-space in order to use it in the <k_operators> code. If we did not need to use it there then the Fourier transform would have been a waste of CPU time. Further, the damping variable used here would have to have been declared as type complex in order to have been able to be Fourier transformed to k-space to be available in the <k_operators> element where it wasn't needed, causing a waste of memory as well.

Another possible function required to define the equations of motion is some integral across the transverse dimensions. These are specified with the <moment_group> element. We do not have one of these in our worked example, but as an example we might need the total number of atoms:

```
<moment_group>
    <moments>number</moments>
    <integrate_dimension>yes</integrate_dimension>
        <![CDATA[
            number += ~phi*phi;
 ]]>
</moment_group>
```

The child elements here name the moments to be defined, and then describe which, if any, of the transverse dimensions are to be integrated. The CDATA block defines the moments themselves. The "number" moment in this example will be integrated over the only transverse dimension. <moment_group> and <functions> elements may be used any number of times in any order. The propagation of the integration equations themselves is performed where the <vectors> tag is placed.

Something else to keep in mind is that when a smooth function is required it is often tempting to use transcendental functions (such as sqrt, exp, sin, cos, log, ...). However, these functions are computationally expensive, particularly on processors that were not designed with such functions in mind. If they are only used once to pre-calculate a constant vector (as is done here) then fine, but if you include them in your main equations then expect a big reduction in performance. If a smooth function that depends on the propagation dimension is required, then, unless it is essential to the model, it is usually better use a low order polynomial approximation.

## 9.7 The filter element

The <filter> element, though not covered in this example, is relatively straightforward. Here is an example which retains a Gaussian profile of the field centered on $t = 0$:

```
<filter>
  <vectors>main</vectors>
  <fourier_space>no</fourier_space>
  <![CDATA[
    phi *= exp(-(t*t));
```

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

As in the <vector> element the space in which the filter is to be applied is specified in a <fourier_space> assignment. <moment_group> and <functions> elements may be used in the <filter> element just as they are used in the <integrate> element.

## 9.8 The output element

```
<output>
  <filename>nlse.xsil</filename>
  <group>
    <sampling>
      <fourier_space> no </fourier_space>
      <lattice>       50 </lattice>
      <moments>pow_dens</moments>
      <![CDATA[
        pow_dens=~phi*phi;
      ]]>
    </sampling>
  </group>
</output>
```

The <output> element is compulsory and defines what will be used as the output.

The <filename> assignment is optional. From version 1.2 **xmds** can produce binary output, as well as the default ascii format. Both output formats will produce an ascii file in XSIL format [2]. The default name for this file is the <simulation> <name> appended with ".xsil". With ascii output all of the data is contained within the XSIL file. However, with binary output, the data is pointed to by the (ascii) XSIL file, where the actual data is distributed among different files labelled according to simulation name and output moment group. In general, the filenames are combined like this: simulation name + mg + moment group number + .dat. For instance, if the simulation name is nlse, then the output binary file for the first moment group will be called nlsemg1.dat.

In order that old output data remains meaningful, **xmds** copies the input simulation script to the name of the output file (overwriting it if it already exists), and then the output data is inserted as <XSIL> elements before the closing </simulation> tag. This way, the input parameters that generated the output data remain known.

Next **xmds** looks for the <group> elements contained within, of which there must be one or more. The <group> element contains moments that are sampled on a common output lattice in a common Fourier space. If necessary the group may then be post-processed after all segments have been performed, for example to calculate moments with the propagation dimension itself in Fourier space.

Thus **xmds** looks for a compulsory <sampling> element within the <group> element that defines how to sample for the moments. It then looks for an optional <post_propagation> element (within the same <group> element) that defines any post-processing required.

Consider firstly the <sampling> element: Firstly, a <fourier_space> assignment is compulsory, which contains as many "yes" or "no" entries as there are transverse dimensions, so that the field may be transformed accordingly prior to sampling. In this example the field's "t" dimension remains in normal space.

Next **xmds** looks for a <lattice> assignment containing the lattice on which to sample on, for to sample the entire field may produce much more data than is necessary to obtain a good plot. In this assignment there should be as many integers as there are transverse dimensions. Each integer should be one of the following values (or else **xmds** will exit with an appropriate error message):

- 0: A lattice integer of zero requests **xmds** to integrate the moments over this dimension. This will cause the output field to no longer be a function of this transverse dimension.

- 1: A lattice integer of 1 requests **xmds** to sample the moments on a cross-sectional slice of this dimension, also causing the output field to loose this transverse dimension. If this dimension is in normal space then **xmds** will extract the slice at the middle lattice point (point number $N/2 + 1$ using integer division), otherwise **xmds** will extract the slice at the zero momentum point, $k = 0$.

- A factor of the main field's corresponding lattice value which is greater than one. In this case, if it is sampled in normal space **xmds** uses a coarser lattice that still spans the entire domain, whereas if it is in Fourier space it samples a narrower window of the k-space domain centered on the zero momentum point, $k = 0$. You may of course specify the same number of lattice points here as there are in this dimension, in which case the output moments are simply mapped point for point.

If there are no transverse dimensions, **xmds** will not look for either of the above two assignments. Next comes the <moments> assignment which lists the names of the moments. Any number of moments are possible, provided there is at least one. Finally the code used for calculating these moments is inserted as the content of the <sampling> element. The moments are complex variables and so the code may be written accordingly.

Next, a <post_propagation> element may optionally follow the <sampling> element. In this element there must be a <fourier_space> element with a "yes" or "no" entry for the propagation dimension, and as many more as there are *remaining* transverse dimensions in the sampled output. In the case of partially collapsed output the correspondence is sequential. For example if the field had three transverse dimensions "x y z" and the"y" dimension was collapsed by using a <lattice> assignment (within the <sampling> element) like "50 1 50", then a <fourier_space> assignment of "yes no yes" would mean transform to Fourier space in the propagation dimension, normal space in the "x" dimension, and Fourier space in the "z" dimension when evaluating the post-processing moments. Finally there is another <moments> assignment (with new names), and more code in the content for processing the moments already derived at the sampling level. As with most other code blocks functions of dimensions may also be included, but only of the dimensions that haven't been collapsed (plus the propagation dimension). Finally, it is only the real part of the <post_propagation> moments that are written as output, or added to the stochastic sums before they are processed.

In the absence of a <post_propagation> element the real parts of the <sampling> moments are used instead.

# 10 More Examples

## 10.1 ndparamp.xmds

```
<?xml version="1.0"?>
<!--Non-Degenerate Parametric Amplifier-->
<!--Simulton formation for logical switching-->

<simulation>

  <name>ndparamp</name>
  <prop_dim>z</prop_dim>
  <error_check>yes</error_check>

  <globals>
    <![CDATA[
      const double e1 =350;
      const double e2 =350;
      const double r1 = 1;
      const double r2 = 1;
      const double vy1 =  0.5;
      const double vy2 = -0.5;
      const double yc1 = -0.2;
      const double yc2 =  0.2;
      const double tc1 = 0;
      const double tc2 = 0;

      double amp1=sqrt(e1/2/M_PI/r1/r1);
      double amp2=sqrt(e2/2/M_PI/r2/r2);
```

```
  ]]>
</globals>

<field>
  <name>main</name>
  <dimensions>   y        t     </dimensions>
  <lattice>     100      100    </lattice>
  <domains>  (-10,10) (-10,10) </domains>
  <samples>1 1</samples>

  <vector>
    <name>main</name>
    <type>complex</type>
    <components>ff1 ff2 sh</components>
    <fourier_space>no no</fourier_space>
    <![CDATA[
      ff1 = pcomplex(amp1*exp(-pow((y - yc1)/r1/2,2)
                      -pow((t - tc1)/r1/2,2)),+vy1*y);
      ff2 = pcomplex(amp2*exp(-pow((y - yc2)/r2/2,2)
                      -pow((t - tc2)/r2/2,2)),+vy2*y);
      sh  = rcomplex(0,0);
    ]]>
  </vector>

  <vector>
    <name>vc1</name>
    <type>double</type>
    <components>damping</components>
    <fourier_space>no no</fourier_space>
    <![CDATA[
      damping=1.0*(1-exp(-pow((y*y + t*t)/8/8,10)));
    ]]>
  </vector>
</field>

<sequence>
  <integrate>
    <algorithm>RK4IP</algorithm>
    <interval>10</interval>
    <lattice>500</lattice>
    <samples>50 50</samples>
    <k_operators>
      <constant>yes</constant>
      <operator_names>Lap1 Lap2</operator_names>
      <![CDATA[
        Lap1 = i*(-(ky*ky + kt*kt) - 1);
        Lap2 = i*(-(ky*ky + kt*kt)/2 - 1);
```

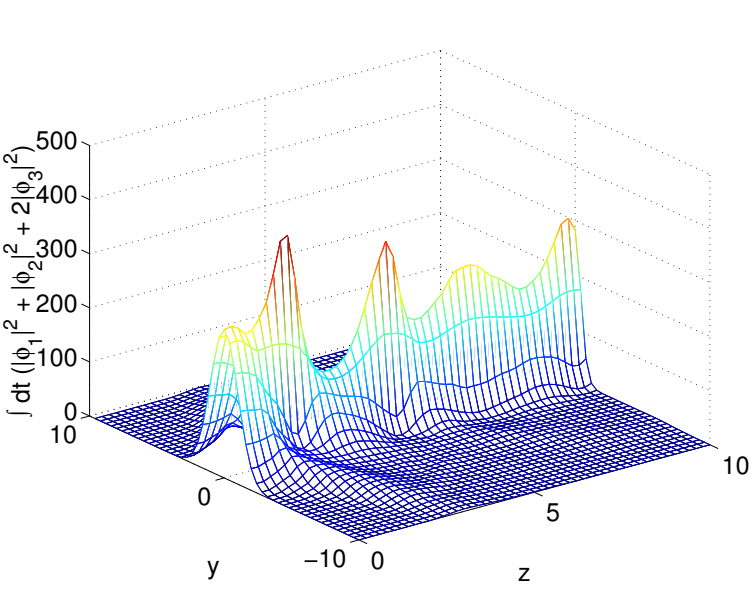
```
        ]]>
      </k_operators>
      <vectors>main vc1</vectors>
        <![CDATA[
          dff1_dz = Lap1[ff1] + i*~ff2*sh  - damping*ff1;
          dff2_dz = Lap1[ff2] + i*~ff1*sh  - damping*ff2;
          dsh_dz  = Lap2[sh]  + i* ff1*ff2 - damping* sh;
        ]]>
    </integrate>
  </sequence>

  <output>

    <group>
      <sampling>
        <fourier_space> no  no </fourier_space>
        <lattice>       50  0  </lattice>
        <moments>pow_dens</moments>
        <![CDATA[
          pow_dens = ~ff1*ff1 + ~ff2*ff2 + 2*~sh*sh;
        ]]>
      </sampling>
    </group>

    <group>
      <sampling>
        <fourier_space> no  no </fourier_space>
        <lattice>       0  0  </lattice>
        <moments>etot</moments>
        <![CDATA[
          etot = ~ff1*ff1 + ~ff2*ff2 + 2*~sh*sh;
        ]]>
      </sampling>
    </group>
  </output>
</simulation>
```

This simulation describes how to solve for the evolution of three bosonic fields governed by a non-degenerate parametric interaction, as described in Equations (10.1) and (10.2). Perhaps not surprisingly, the majority of the above script is to generate the particular initialisation conditions and to define a number of output moments—the portion concerned with actually implementing these equations is not that great.

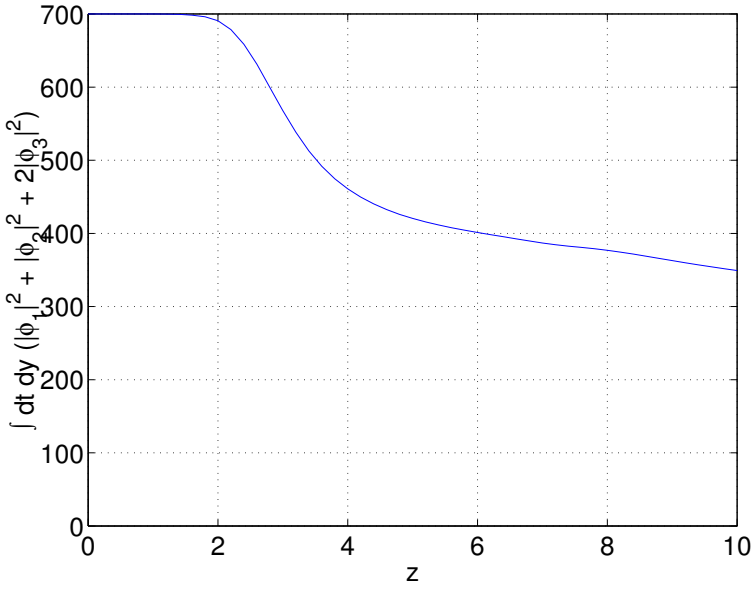
$$\frac{\partial \phi_j}{\partial \xi} = i\left[\left(\frac{\partial^2}{\partial \tau^2} + \frac{\partial^2}{\partial \zeta^2} + i\Gamma(\tau,\zeta) - 1\right)\phi_j + \phi_{3-j}^{*}\phi_3\right], \quad j = 1,2; \tag{10.1}$$

$$\frac{\partial \phi_3}{\partial \xi} = i\left[\left(\frac{1}{\sigma}\frac{\partial^2}{\partial \tau^2} + \frac{1}{2}\frac{\partial^2}{\partial \zeta^2} + i\Gamma(\tau,\zeta) - \gamma\right)\phi_3 + \phi_1\phi_2\right]. \tag{10.2}$$

The main difference between this simulation and the `nlse.xmds` simulation is that it now has a three component field which has two transverse dimensions. Also two moment groups are being evaluated, one being integrated over the "`t`" dimension, and the other integrated over both transverse dimensions.




a) Time integrated intensity




b) Total energy (vs z)

FIGURE 10.1: Results for ndparamp.xmds

## 10.2 kubo.xmds

```
<?xml version="1.0"?>
<!--Example Kubo oscillator simulation-->

<simulation>

  <name>kubo</name>
  <prop_dim>t</prop_dim>
  <error_check>yes</error_check>
  <stochastic>yes</stochastic>
  <paths>1</paths>
  <use_mpi>no</use_mpi>
  <seed>1 2</seed>
  <noises>1</noises>

  <field>
    <samples>1</samples>
    <vector>
      <name>main</name>
```

```
      <type>complex</type>
      <components>z</components>
      <![CDATA[
        z = 1;
      ]]>
    </vector>
  </field>

  <sequence>
    <integrate>
      <algorithm>SIEX</algorithm>
      <interval>10</interval>
      <lattice>1000</lattice>
      <samples>100</samples>
      <iterations>3</iterations>
      <![CDATA[
        dz_dt = i*z*n_1;
      ]]>
    </integrate>

  </sequence>

  <output>
    <group>
      <sampling>
      <moments>realz</moments>
      <![CDATA[
        realz = z;
      ]]>
      </sampling>
    </group>
  </output>
</simulation>
```

The kubo oscillator is described in Equation (10.3), in which the argument of the complex vector $z$ is "blown" about by a (real) Gaussian noise term, $\xi(t)$. This is a simple stochastic ODE.

$$\frac{\partial z}{\partial t} = i\xi(t)z. \tag{10.3}$$

Such Gaussian noise terms, in analytic form, are correlated in time and space through Dirac delta functions, as shown in Equation (10.4).

$$\langle \xi_i(\mathbf{x})\xi_j(\mathbf{x}')\rangle = \delta_{i,j}\Pi_{i=0}^{N}\delta(x^i - x'^i). \tag{10.4}$$

However, when solving stochastic DEs numerically, algorithms work with discrete time intervals and lattice spacings. Therefore these Dirac delta correlations must be transformed to Kronecker delta correlations using the integration time step and the spatial volume of the

lattice, as shown in Equation (10.5):

$$\langle \xi_i(\mathbf{x}) \xi_j(\mathbf{x}') \rangle = \frac{\delta_{i,j} \Pi_{i=0}^{N} \delta_{x^i, x'^i}}{\Pi_{i=0}^{N} \Delta x^i}. \tag{10.5}$$

The good news is that **xmds** calculates this for the user—all that has to be done is to specify, as a simple `<noises>` assignment within the `<simulation>` element, the maximum number of noise terms required in any one segment, and then reference them as `n_1`, `n_2`, etc. as can be seen in this example. These noises are available within the initialisation code for each field `<vector>`, within the main integration equations code (not in the `<k_operators>` code), and in the code for any `<filter>` segments. Within the initialisation code and `<filter>` code the variances of the noises are determined by the lattice cell volume product for the particular `<fourier_space>` specification as shown in Equation (10.6):

$$\langle n_i n_j \rangle = \frac{\delta_{i,j}}{\Pi \Delta k^m \Pi \Delta x^n}, \tag{10.6}$$

where $m$ are the transverse dimensions in Fourier space and $n$ the transverse dimensions in normal space. Also note that

$$\Delta k^i = \frac{2\pi}{x^i_{max} - x^i_{min}}. \tag{10.7}$$

Within the main integration equations the variances must also reflect the integration step size, as given by Equation (10.8):

$$\langle n_i n_j \rangle = \frac{\delta_{i,j}}{\Pi_{i=0}^{N} \Delta x^i}. \tag{10.8}$$

**xmds** uses the Box-Mueller technique, as shown in Equation (10.9), which generates a pair of Gaussian noises, $\xi_1$ and $\xi_2$, from a pair of random numbers, $x_1$ and $x_2$, that have a uniform distribution between zero and one.

$$\xi_1 + i\xi_2 = [-2\Delta ln(x_1)]^{\frac{1}{2}} e^{i2\pi x_2} \; ; \; \mathcal{P}(x_i < y) = y \; ; \; y : 0 < y \leq 1; y \epsilon \mathcal{R}. \tag{10.9}$$

Estimating the error between full and half step sizes now poses an interesting problem, since both evolutions must use the same underlying noise (which is a function of both space and time) if the difference between these paths is to be meaningful. The random number generator must be reset before each of these integrations, but how is the noise in the half-step case appropriated? There are two methods. The first is to use the same noise for both half-steps as is used for the one whole step. This is undesirable since it makes sense to use the half-step integration results (being the more accurate) for the final output, and therefore independent noises must be used for each step. So, the second solution is to do just this, and use the average of the two noises when calculating the full-step integration. Now, suppose the problem uses $N$ noises and has a transverse lattice of $M$ points. Within the SIEX, RK4IP, and the RK4EX integration algorithms the main field vector is swept through severals times in the course of each time step, thus a $M \times N$ vector of noise must be calculated at the beginning of the time step and referenced during the calculation of the main field vector's derivatives. In the full-step case two such vectors are calculated, and

then averaged to provide the equivalent full-step noise. However, the SIIP algorithm only sweeps though the main field vector once for each time step, and so only $N$ noises need be calculated at a time (thus saving on memory and RAM access), provided *two* independent random number generators are used: the first generator is used for the first half step, the second for the second half step, and the average of both is used for the full step. In C this is done with the *erand48(n)* function which uses the 48-bit integer $n$ to generate the next random number, advancing $n$ to the next in sequence in the process. The states of the two independent generators are simply two independent integers, $n_1$ and $n_2$. The user supplies the initial values for these integers in the `<seed>` assignment.

Stochastic problems are very well suited to parallel computer architectures, as different paths can run on different processors, and do not have to transfer information until the integration is complete. Multiple path stochastic problems such as this may be parallised using MPI routines. If an MPI compiler was specified in the configuration step of installation (this will often be `mpicc`), then all that needs to be done is to toggle the optional `<use_mpi>` assignment to `yes`. **xmds** will then place the appropriate MPI calls in the output code and compile it with the MPI compiler. The executable should then be run through the MPI execution handler (probably `mpirun`), with the number of processors option supplied. For example if 16 processors are available then the final command for execution would be

```
% mpirun -np 16 kubo
```

Note that whole paths are assigned independently to the processors, so there is no benefit in specifying more processors than there are paths in the simulation. It is not necessary for the number of processors to be a factor of the number of paths, some processors will simply do one more paths than others.

Also note that for both of the semi-implicit algorithms, SIEX and SIIP, an `<iterations>` assignment may be used within the `<integrate>` element to specify the number of iterations to use in the method (refer Sections 6.3.3, 6.4.3, and 6.4.9). This assignment is optional, and will default to three when absent.

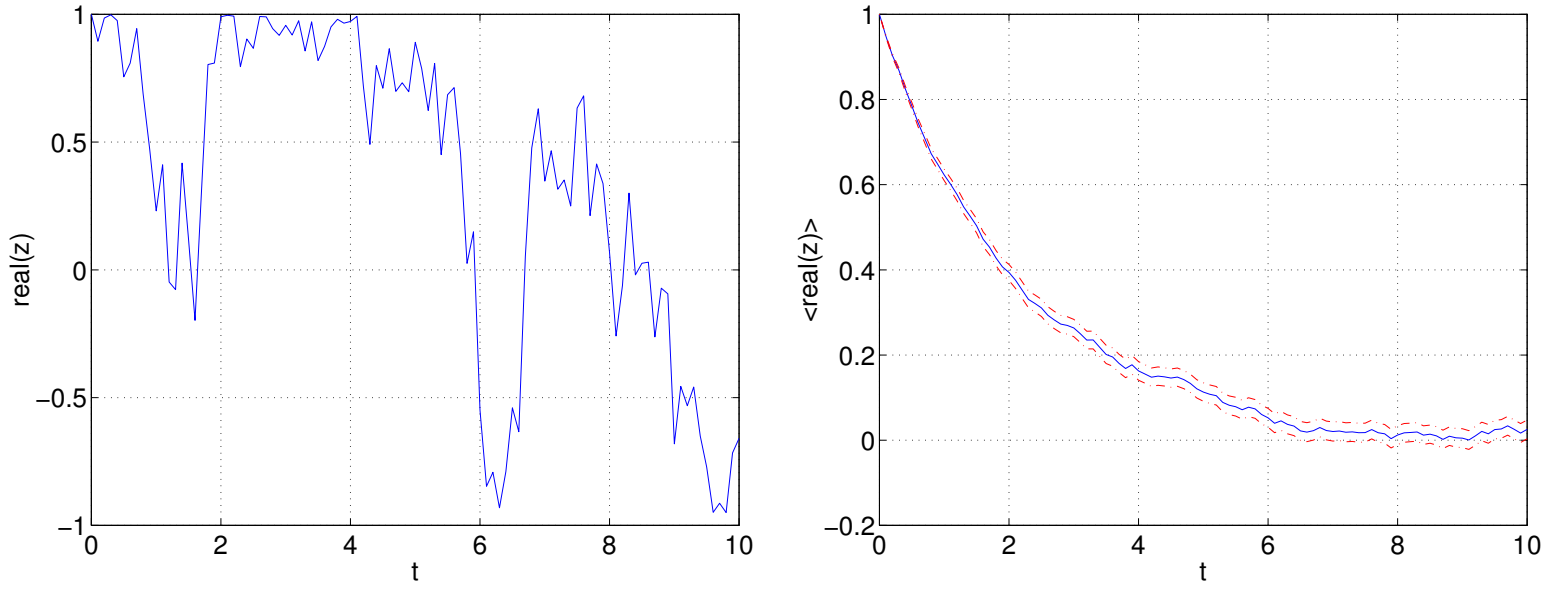
The reason that the kubo oscillator is used as an example is that it has an analytic solution, as shown in Equation (10.10). Figure 10.2 shows the results for a single trajectory and an averaged trajectory, illustrating the expected behaviour.

$$\langle z(t) \rangle = z_0 e^{-\frac{t}{2}}. \tag{10.10}$$

Since the variance of the noise terms scale with the inverse of the integration step size, the integration method suffers a loss of order with regard to error vs step size. While this is normally a second order method for non-stochastic problems, it becomes a first order method for problems with noise, as was explained in Section 6.3.1.

## 10.3 fibre.xmds

```
<?xml version="1.0"?>
<!--Example fibre noise simulation-->

<simulation>
```




a) Single path b) 1024 path mean

FIGURE 10.2: Results for kubo.xmds

```
  <name>fibre</name>
  <prop_dim>t</prop_dim>
  <error_check>yes</error_check>
  <stochastic>yes</stochastic>
  <use_mpi>no</use_mpi>
  <paths>1</paths>
  <seed>1 2</seed>
  <noises>2</noises>

  <globals>
<![CDATA[
const double ggamma = 1;
const double beta = sqrt(2*2*M_PI*ggamma/10);
]]>
  </globals>

  <field>
    <name>main</name>
    <dimensions>   x   </dimensions>
    <lattice>     50  </lattice>
    <domains>  (-5,5)  </domains>
    <samples>1</samples>

    <vector>
      <name>main</name>
      <type>complex</type>
      <components>phi</components>
```

```
      <fourier_space>no</fourier_space>
<![CDATA[
phi=0;
]]>
    </vector>
  </field>

  <sequence>
    <integrate>
      <algorithm>SIIP</algorithm>
      <interval>2.5</interval>
      <lattice>5000</lattice>
      <samples>50</samples>
      <k_operators>
        <constant>yes</constant>
        <operator_names>L</operator_names>
<![CDATA[
L = i*(-kx*kx);
]]>
      </k_operators>
      <iterations>3</iterations>
<![CDATA[
dphi_dt = L[phi] - ggamma*phi + beta/sqrt(2)*complex(n_1,n_2);
]]>
    </integrate>
  </sequence>

  <output>
    <group>
      <sampling>
      <fourier_space> yes </fourier_space>
      <lattice>       50  </lattice>
      <moments>pow_dens</moments>
<![CDATA[
pow_dens = conj(phi)*phi;
]]>
      </sampling>
    </group>
  </output>
</simulation>
```

This simulation solves Equation (10.11), in which a one dimensional damped field is subject to a complex noise. This is a stochastic PDE.

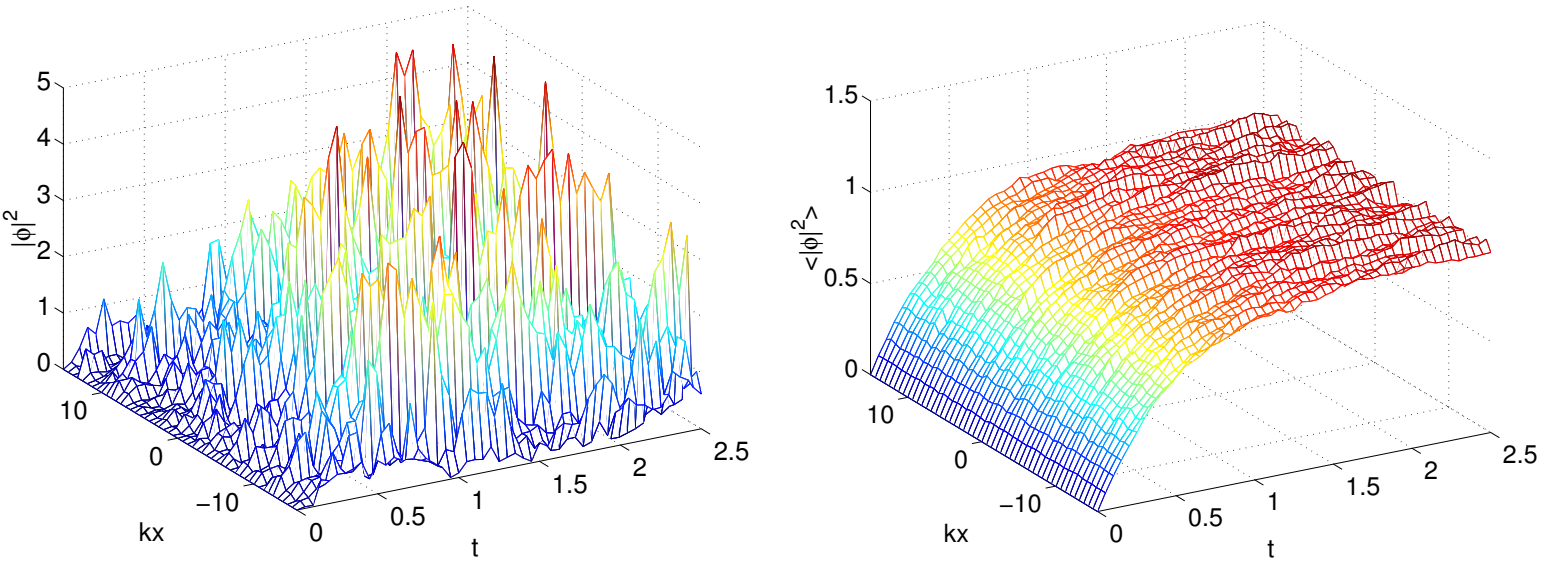
$$\frac{\partial \psi}{\partial t} = -i\frac{\partial^2 \psi}{\partial x^2} - \gamma\psi + \frac{\beta}{\sqrt{2}}(\xi_1(x,t) + i\xi_2(x,t)). \tag{10.11}$$

Again the reason for using this as an example of a stochastic PDE is that it has an analytic solution, as shown in Equation (10.12). Figure 10.3 displays the results of this

simulation in Fourier space for a single trajectory and an averaged trajectory, which appear as expected.

$$\left\langle |\psi(k,t)|^2 \right\rangle = e^{-2\gamma t}|\psi_0(k)|^2 + \frac{\beta^2 L_x}{4\pi\gamma}(1 - e^{-2\gamma t}), \tag{10.12}$$

where $L_x$ is the length of the $x$ domain.




a) Single path b) 1024 path mean

FIGURE 10.3: Results for fibre.xmds

One important issue here is that the variance of the noise terms now scales with the product of the number of lattice points (for a given domain). Hence changing to a finer lattice actually increases the single trajectory error, and the relationship between the error and the lattice product will depend on the order of the spatial derivatives. The only way to overcome this is to reduce the integration step size, which added to the fact that there are more lattice points in the first place, means that fine lattice resolution in a stochastic PDE is computationally *very* expensive.

## 10.4 tla.xmds

```
<?xml version="1.0"?>
<!--Two Level Atom Example simulation to illustrate a
cross propagating field-->

<simulation>

  <prop_dim> z </prop_dim>

  <globals>
```

```
  <![CDATA[
    const double g  = 1;
    const double t0 = 1;
  ]]>
</globals>

<field>
  <dimensions>   t    </dimensions>
  <lattice>     100   </lattice>
  <domains> (-10, 15) </domains>
  <samples> 1 0 </samples>

  <vector>
    <name> main </name>
    <type> double </type>
    <components> E </components>
    <![CDATA[
      E = 2/t0/cosh(t/t0);
    ]]>
  </vector>

  <vector>
    <name> cross </name>
    <type> double </type>
    <components> P N </components>
    <![CDATA[
        P =  0;
        N = -1;
    ]]>
  </vector>
</field>

<sequence>
  <integrate>
    <algorithm> RK4EX </algorithm>
    <interval> 4 </interval>
    <lattice> 50 </lattice>
    <samples> 50 50 </samples>
    <vectors> main cross </vectors>
      <![CDATA[
        dE_dz =  g*P;
      ]]>
    <cross_propagation>
      <vectors> cross </vectors>
      <prop_dim> t </prop_dim>
      <![CDATA[
        dP_dt =  E*N;
```

```
          dN_dt = -E*P;
        ]]>
      </cross_propagation>
    </integrate>
  </sequence>

  <output>

    <group>
      <sampling>
        <lattice> 50 </lattice>
        <moments> pow_dens </moments>
        <![CDATA[
          pow_dens = E*E;
        ]]>
      </sampling>
    </group>

    <group>
      <sampling>
        <vectors> main cross </vectors>
        <lattice> 50 </lattice>
        <moments> P_out N_out </moments>
        <![CDATA[
          P_out = P;
          N_out = N;
        ]]>
      </sampling>
      </group>
  </output>
</simulation>
```

This simulation solves for the propagation of an optical pulse through a field of atoms having a transition frequency tuned to that of the optical pulse centre frequency. The atoms are modelled as "two level" atoms. The propagation equations, shown in Equation (10.13), are deceptively simple.

$$\begin{aligned} \frac{\partial E(t,z)}{\partial z} &= gP, \\ \frac{\partial P(t,z)}{\partial t} &= EN, \\ \frac{\partial N(t,z)}{\partial t} &= -EP. \end{aligned} \tag{10.13}$$

The reality of this problem is that there are three components, two propagating in the main propagation dimension, $t$, and the other in the transverse dimension $z$. The component $E$ is the electric field amplitude, $P$ the polarisation state of the atoms, and $N$ the excitation state of the atoms (-1 being all in the ground state and +1 being all in the excited state).

Lastly $g$ is a coupling constant between the electric field and the atoms.

The curious feature of this set of PDEs, in fact the very reason why it is chosen it as an example, is that there exists a soliton solution for the electric field, as shown in Equation (10.14):

$$E(t,z) = \frac{2}{gt_0}\text{sech}\left(\frac{t-az}{t_0}\right), \tag{10.14}$$

which is time lagged with propagation at the rate

$$a = \frac{g}{(t_0)^2}. \tag{10.15}$$

The result for the electric field in the above simulation is shown in Figure 10.4, in which the soliton solution is evident.

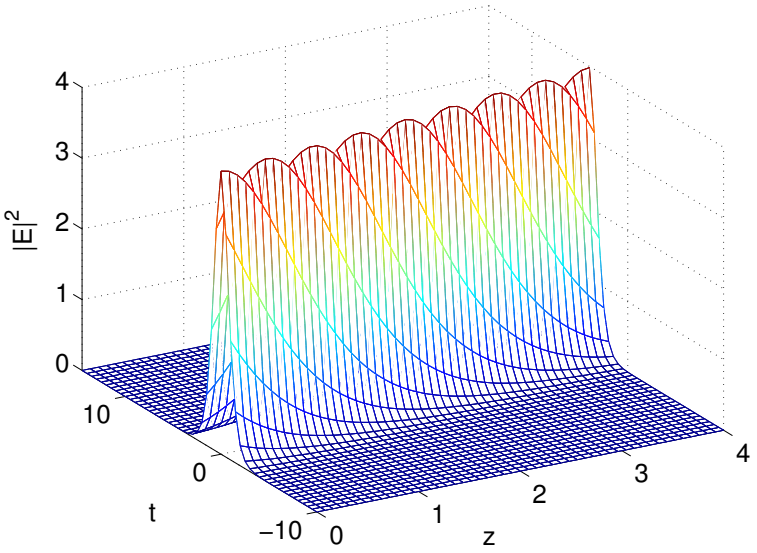



FIGURE 10.4: Results for tla.xmds

The cross propagating components, $N$ and $P$, may still be thought of as existing in the same space as the main vector component, $E$, and so they are declared as an extra vector in the `<field>` element. However, the equations governing the evolution of such cross vectors are not allowed to include any transverse derivatives – i.e. they are not allowed to be PDEs. Therefore, through the main vector equations may be PDEs, the cross propagating vector need not be transformed to Fourier space when the main vector is. So although the cross vector components could be included as part of the main vector, they are better defined as a separate vector for efficiency reasons.

In the SIIP integration algorithm the transverse evolution of the cross vector is calculated simultaneously with the forward evolution of the main vector, but in all other algorithms the cross vector is calculated prior to calculating the main vector derivatives. Thus the governing equations for the cross vector must be separated from those for the main vector. This is done by including a `<cross_propagation>` element within the main `<integrate>` element, and placing the cross vector equations within. Also required within this element

is a list of the <vectors> that are to be cross propagated, and a <prop_dim> assignment specifying the dimension of cross propagation. The <vectors> that were made accessible for the main equations will also be accessible here.

## 10.5 highdim.xmds

```
<?xml version="1.0"?>
<simulation>
  <name>highdim</name>

  <!-- Global system parameters and functionality -->
  <prop_dim>t</prop_dim>
  <error_check>yes</error_check>
  <use_mpi>yes</use_mpi>
  <use_wisdom>yes</use_wisdom>
  <benchmark>yes</benchmark>

  <!-- Global variables for the simulation -->
  <globals>
  <![CDATA[
    const double noise = 0.0;
    const double hbar = 1.05500000000e-34;
    const double M = 1.409539200000000e-25;
    const double omegax = 0.58976353090742;
    const double omegay = 0.58976353090742;
    const double omegaz = 0.58976353090742/30;
    const double U11 = 2.974797272874263e-51;
    const double U13 = -1.417820412490823e-50;
    const double U33 = 2.974797272874263e-51;
    const double inum = 1.0e6;
    const double Uoh11 = U11/hbar;
    const double Uoh13 = U13/hbar;
    const double Uoh33 = U33/hbar;
    const double mu = pow(15*inum*U11*omegax*omegay
              *omegaz/M_PI/4,0.4)*pow(M,0.6)/2;
    const double delta = 1.0e9;
    const double F = 2.0e-2;
    const double g = sqrt(Uoh11*2.0/delta);
    const double loss11=1.0e-2;
    const double loss12=1.6e-22;
    const double loss31=1.0e-2;
    const double loss32=1.6e-22;
    const double loss132=8.0e-17;
    const double chi = F*g*delta;
    const double biggamma = g*g*delta/2;
    const double gam13 = Uoh13/chi;
```

```
    const double gam33 = Uoh33/chi;
    const double gameff = (Uoh11-biggamma)/chi;
    const double gamloss11=loss11/2/chi;
    const double gamloss12=loss12/chi;
    const double gamloss31=loss31/2/chi;
    const double gamloss32=loss32/chi;
    const double gamloss132=loss132/chi;
    const double cnoise =  noise/sqrt(2.0);
  ]]>
  </globals>

    <argv>
            <arg>
                <name>kjoek</name>
                <type>double</type>
                <default_value>-1.0e6</default_value>
            </arg>
            <arg>
                <name>joekappamax</name>
                <type>double</type>
                <default_value>1.0e2</default_value>
            </arg>
    </argv>

  <!-- Field to be integrated over -->
  <field>
    <dimensions>x y z</dimensions>
    <lattice>16 16 16</lattice>
    <domains>(-1.2e-4,1.2e-4) (-1.2e-4,1.2e-4) (-8.0e-3,8.0e-3)</domains>
    <samples>1 1 1</samples>

    <vector>
      <name> vc1 </name>
      <type>double</type>
      <components>vcore V1r V3r gV1r gV3r</components>
      <fourier_space>no no no</fourier_space>
      <![CDATA[
   vcore = (omegax*omegax*x*x+omegay*omegay*y*y+omegaz*omegaz*z*z);
   V1r = 0.5*M*vcore/hbar/chi -(gameff+gam13/2)/2/(dx*dy*dz);
   V3r = M*vcore/hbar/chi -(gam13/2+gam33)/2/(dx*dy*dz);
   gV1r = 0.5*M*vcore/hbar/chi;
   gV3r = M*vcore/hbar/chi;
      ]]>
    </vector>

    <vector>
      <name> main </name>
```

```
    <type>complex</type>
    <components>phi1a phi1b phi3a phi3b gphi1a gphi3a</components>
    <fourier_space>no no no</fourier_space>
    <vectors> vc1 </vectors>
    <![CDATA[
      const double realfn = (mu-0.5*M*vcore)/Uoh11/hbar;

      phi1a = realfn>0. ? complex(sqrt(realfn),0) : complex(0,0);
      phi1b = realfn>0. ? complex(sqrt(realfn),0) : complex(0,0);
      phi3a = complex(0,0);
      phi3b = complex(0,0);
      gphi1a = realfn>0. ? complex(sqrt(realfn),0) : complex(0,0);
      gphi3a = complex(0,0);
    ]]>
  </vector>
</field>

<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm>ARK89IP</algorithm>
    <interval>1e-7</interval>
     <tolerance>1.0e-7</tolerance>
    <lattice>1000</lattice>
    <samples>10 10 1</samples>
    <k_operators>
      <constant>yes</constant>
      <operator_names> L2p L2n L4p L4n </operator_names>
      <![CDATA[
        L2p = complex(0,-hbar/M/2/chi*(kx*kx+ky*ky+kz*kz));
        L2n = complex(0, hbar/M/2/chi*(kx*kx+ky*ky+kz*kz));
        L4p = complex(0,-hbar/M/4/chi*(kx*kx+ky*ky+kz*kz));
        L4n = complex(0, hbar/M/4/chi*(kx*kx+ky*ky+kz*kz));
      ]]>
    </k_operators>

      <moment_group>
      <moments>chippy</moments>
      <integrate_dimension>yes yes yes</integrate_dimension>
          <![CDATA[
              chippy += ~gphi1a*gphi1a;
              ]]>
      </moment_group>

      <moment_group>
      <moments>ippy ichippy</moments>
      <integrate_dimension>no no no</integrate_dimension>
```

```
            <![CDATA[
                ippy += phi1a;
                ichippy += gphi1a;
                ]]>
        </moment_group>
        <moment_group>
        <moments>py ic</moments>
        <integrate_dimension>yes yes no</integrate_dimension>
            <![CDATA[
                py += phi1a;
                ic += gphi1a;
                ]]>
        </moment_group>
        <moment_group>
        <moments>ppy ichi</moments>
        <integrate_dimension>no yes yes</integrate_dimension>
            <![CDATA[
                ppy += phi1a;
                ichi += gphi1a;
                ]]>
        </moment_group>

      <vectors> main vc1 </vectors>
      <![CDATA[
        const complex dens1 = phi1b*phi1a;
        const complex dens3 = phi3b*phi3a;

    const double gdens1 = (gphi1a.re*gphi1a.re+gphi1a.im*gphi1a.im);
    const double gdens3 = (gphi3a.re*gphi3a.re+gphi3a.im*gphi3a.im);

        dphi1a_dt = L2p[phi1a] + (-i*V1r-gamloss11
          +(gamloss132/2+gamloss12)/2/(dx*dy*dz))*phi1a
          + (-i*gameff-gamloss12)*dens1*phi1a
          - (i*gam13+gamloss132)*dens3*phi1a -i*phi1b*phi3a
          + i*chippy*ippy;

        dphi1b_dt = L2n[phi1b] + (i*V1r-gamloss11
          +(gamloss132/2+gamloss12)/2/(dx*dy*dz))*phi1b
          + (i*gameff-gamloss12)*dens1*phi1b
          + (i*gam13-gamloss132)*dens3*phi1b +i*phi1a*phi3b;

        dphi3a_dt = L4p[phi3a] + (-i*V3r-gamloss31
          +(gamloss132/2+gamloss32)/2/(dx*dy*dz))*phi3a
          + (-i*gam33-gamloss32)*dens3*phi3a
          - i*0.5*phi1a*phi1a -(i*gam13+gamloss132)*dens1*phi3a;

        dphi3b_dt = L4n[phi3b] + (i*V3r-gamloss31
```

```
        +(gamloss132/2+gamloss32)/2/(dx*dy*dz))*phi3b
        + (i*gam33-gamloss32)*dens3*phi3b
        + i*0.5*phi1b*phi1b +(i*gam13-gamloss132)*dens1*phi3b;

      dgphi1a_dt = L2p[gphi1a] + (-i*gV1r-gamloss11)*gphi1a
         +(-i*gameff-gamloss12)*gdens1*ichippy
        - (i*gam13+gamloss132)*gdens3*gphi1a-i*conj(gphi1a)*gphi3a;

      dgphi3a_dt = L4p[gphi3a] + (-i*gV3r-gamloss31)*gphi3a
        +(-i*gam33-gamloss32)*gdens3*gphi3a
        - i*0.5*gphi1a*gphi1a +(i*gam13-gamloss132)*gdens1*gphi3a;
    ]]>
  </integrate>
</sequence>

<!-- The output to generate -->
<output format="binary" precision="double">
  <group>
    <sampling>
      <fourier_space>   no    no no</fourier_space>
      <lattice>         16    1  1</lattice>
      <moments>atoms molecules gatoms gmolecules</moments>
      <![CDATA[
        atoms=phi1b*phi1a;
        molecules=phi3b*phi3a;
        gatoms=conj(gphi1a)*gphi1a;
        gmolecules=conj(gphi3a)*gphi3a;
      ]]>
    </sampling>
  </group>
  <group>
    <sampling>
      <fourier_space>    no   no  no</fourier_space>
      <lattice>           0    16   0</lattice>
      <moments>rn_1 rn_2 grn_1 grn_2 excitedn</moments>
      <![CDATA[
  rn_1 = phi1b*phi1a;
  rn_2 = phi3b*phi3a;
  grn_1 = conj(gphi1a)*gphi1a;
  grn_2 = conj(gphi3a)*gphi3a;
  excitedn = g*g/4*phi1b*phi1b*phi1a*phi1a+F*F*phi3b*phi3a
                  - F*g/2*(phi1b*phi1b*phi3a+phi1a*phi1a*phi3b);
      ]]>
    </sampling>
  </group>
  <group>
    <sampling>
```

```
        <fourier_space>   no no no</fourier_space>
        <lattice>         4    8  16</lattice>
        <moments>atomsr moleculesr atomsi moleculesi</moments>
        <![CDATA[
          atomsr=phi1a;
          moleculesr=phi3a;
          atomsi=-i*gphi1a;
          moleculesi=-i*gphi3a;
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

This simulation is included to highlight the usage of moment groups in evolution. Here we wish to use various variables integrated over one, two or all the transverse dimensions. This is done in integrate or filter elements by the inclusion of a moment group. Using the first as an example:

```
        <moment_group>
        <moments>chippy</moments>
        <integrate_dimension>yes yes yes</integrate_dimension>
    <![CDATA[
chippy += ~gphi1a*gphi1a;
]]>
        </moment_group>
```

The syntax is similar to output and filter syntax, but note that the equality must be a "+=" and not simply "=". This element provides the integral of the modulus squared of the field gphi1a, which can then be used normally in the integration code by the designated name, chippy. Other variables are actually fields in which any number of transverse dimensions may be integrated, and the others are left.

When `<functions>` and `<moment_group>` elements are used, the position of the `<vectors>` tag is crucial. It specifies when the integrate code is to be executed, which will usually need to be after the moment groups are calculated.

## 10.6 highdim_vector_version.xmds

```
<?xml version="1.0"?>
<simulation>
  <name>highdim</name>
  <!-- Global system parameters and functionality -->
  <prop_dim>t</prop_dim>
  <error_check>yes</error_check>
  <use_mpi>yes</use_mpi>
  <use_wisdom>yes</use_wisdom>
```

```
<benchmark>yes</benchmark>

<!-- Global variables for the simulation -->
<globals>
<![CDATA[
  const double noise = 0.0;
  const double hbar = 1.05500000000e-34;
  const double M = 1.409539200000000e-25;
  const double omegax = 0.58976353090742;
  const double omegay = 0.58976353090742;
  const double omegaz = 0.58976353090742/30;
  const double U11 = 2.974797272874263e-51;
  const double U13 = -1.417820412490823e-50;
  const double U33 = 2.974797272874263e-51;
  const double inum = 1.0e6;
  const double Uoh11 = U11/hbar;
  const double Uoh13 = U13/hbar;
  const double Uoh33 = U33/hbar;
  const double mu = pow(15*inum*U11*omegax*omegay
          *omegaz/M_PI/4,0.4)*pow(M,0.6)/2;
  const double delta = 1.0e9;
  const double F = 2.0e-2;
  const double g = sqrt(Uoh11*2.0/delta);
  const double loss11=1.0e-2;
  const double loss12=1.6e-22;
  const double loss31=1.0e-2;
  const double loss32=1.6e-22;
  const double loss132=8.0e-17;
  const double chi = F*g*delta;
  const double biggamma = g*g*delta/2;
  const double gam13 = Uoh13/chi;
  const double gam33 = Uoh33/chi;
  const double gameff = (Uoh11-biggamma)/chi;
  const double gamloss11=loss11/2/chi;
  const double gamloss12=loss12/chi;
  const double gamloss31=loss31/2/chi;
  const double gamloss32=loss32/chi;
  const double gamloss132=loss132/chi;
  const double cnoise =  noise/sqrt(2.0);
]]>
</globals>

  <argv>
        <arg>
            <name>kjoek</name>
            <type>double</type>
            <default_value>-1.0e6</default_value>
```

```
            </arg>
            <arg>
                <name>joekappamax</name>
                <type>double</type>
                <default_value>1.0e2</default_value>
            </arg>
    </argv>

  <!-- Field to be integrated over -->
  <field>
    <dimensions>x y z</dimensions>
    <lattice>16 16 16</lattice>
<domains>(-1.2e-4,1.2e-4) (-1.2e-4,1.2e-4) (-8.0e-3,8.0e-3)</domains>
    <samples>1 1 1</samples>

    <vector>
      <name> vc1 </name>
      <type>double</type>
      <components>Vr(5)</components>
      <fourier_space>no no no</fourier_space>
      <![CDATA[
     Vr(1) = (omegax*omegax*x*x+omegay*omegay*y*y+omegaz*omegaz*z*z);
     Vr(2) = 0.5*M*Vr(1)/hbar/chi -(gameff+gam13/2)/2/(dx*dy*dz);
     Vr(3) = M*Vr(1)/hbar/chi -(gam13/2+gam33)/2/(dx*dy*dz);
     Vr(4) = 0.5*M*Vr(1)/hbar/chi;
     Vr(5) = M*Vr(1)/hbar/chi;
      ]]>
    </vector>

    <vector>
      <name> main </name>
      <type>complex</type>
      <components> phi(6) </components>
      <fourier_space>no no no</fourier_space>
      <vectors> vc1 </vectors>
      <![CDATA[
        const double realfn = (mu-0.5*M*Vr(1))/Uoh11/hbar;

       for(long j=1; j<7; j++) {
         if (j==1||j==2||j==5)
          phi(j) = realfn>0. ? complex(sqrt(realfn),0) : complex(0,0);
        else
          phi(j) = complex(0,0);
        }
      ]]>
    </vector>
  </field>
```

```
<!-- The sequence of integrations to perform -->
<sequence>
  <integrate>
    <algorithm>ARK89EX</algorithm>
    <interval>1e-7</interval>
    <tolerance>1.0e-7</tolerance>
    <lattice>1000</lattice>
    <samples>10 10 1</samples>
    <k_operators>
      <constant>yes</constant>
      <operator_names> L2p L2n L4p L4n </operator_names>
      <![CDATA[
        L2p = complex(0,-hbar/M/2/chi*(kx*kx+ky*ky+kz*kz));
        L2n = complex(0, hbar/M/2/chi*(kx*kx+ky*ky+kz*kz));
        L4p = complex(0,-hbar/M/4/chi*(kx*kx+ky*ky+kz*kz));
        L4n = complex(0, hbar/M/4/chi*(kx*kx+ky*ky+kz*kz));
      ]]>
    </k_operators>

      <moment_group>
      <moments>chippy</moments>
      <integrate_dimension>yes yes yes</integrate_dimension>
          <![CDATA[
              chippy += ~phi(5)*phi(5);
              ]]>
      </moment_group>

      <moment_group>
      <moments>ippy ichippy</moments>
      <integrate_dimension>no no no</integrate_dimension>
          <![CDATA[
              ippy += phi(1);
              ichippy += phi(5);
              ]]>
      </moment_group>
      <moment_group>
      <moments>py ic</moments>
      <integrate_dimension>yes yes no</integrate_dimension>
          <![CDATA[
              py += phi(1);
              ic += phi(5);
              ]]>
      </moment_group>
      <moment_group>
      <moments>ppy ichi</moments>
      <integrate_dimension>no yes yes</integrate_dimension>
```

```
            <![CDATA[
                ppy += phi(1);
                ichi += phi(5);
                ]]>
        </moment_group>

     <vectors> main vc1 </vectors>
     <![CDATA[
        const complex dens1 = phi(2)*phi(1);
        const complex dens3 = phi(4)*phi(3);

    const double gdens1 = (phi(5).re*phi(5).re+phi(5).im*phi(5).im);
    const double gdens3 = (phi(6).re*phi(6).re+phi(6).im*phi(6).im);

        dphi_dt(1) = L2p[phi](1) + (-i*Vr(2)-gamloss11
             +(gamloss132/2+gamloss12)/2/(dx*dy*dz))*phi(1)
             + (-i*gameff-gamloss12)*dens1*phi(1)
             - (i*gam13+gamloss132)*dens3*phi(1) -i*phi(2)*phi(3)
             + i*chippy*ippy;

        dphi_dt(2) = L2n[phi](2) + (i*Vr(2)-gamloss11
             +(gamloss132/2+gamloss12)/2/(dx*dy*dz))*phi(2)
             + (i*gameff-gamloss12)*dens1*phi(2)
             + (i*gam13-gamloss132)*dens3*phi(2) +i*phi(1)*phi(4);

        dphi_dt(3) = L4p[phi](3) + (-i*Vr(3)-gamloss31
             +(gamloss132/2+gamloss32)/2/(dx*dy*dz))*phi(3)
             + (-i*gam33-gamloss32)*dens3*phi(3)
             - i*0.5*phi(1)*phi(1) -(i*gam13+gamloss132)*dens1*phi(3);

        dphi_dt(4) = L4n[phi](4) + (i*Vr(3)-gamloss31+(gamloss132/2
           +gamloss32)/2/(dx*dy*dz))*phi(4)
           + (i*gam33-gamloss32)*dens3*phi(4)
           + i*0.5*phi(2)*phi(2) +(i*gam13-gamloss132)*dens1*phi(4);

        dphi_dt(5) = L2p[phi](5) + (-i*Vr(4)-gamloss11)*phi(5)
           +(-i*gameff-gamloss12)*gdens1*ichippy
           - (i*gam13+gamloss132)*gdens3*phi(5)-i*conj(phi(5))*phi(6);

        dphi_dt(6) = L4p[phi](6) + (-i*Vr(5)-gamloss31)*phi(6)
           +(-i*gam33-gamloss32)*gdens3*phi(6)
           - i*0.5*phi(5)*phi(5) +(i*gam13-gamloss132)*gdens1*phi(6);
     ]]>
   </integrate>
</sequence>

<!-- The output to generate -->
```

```
  <output format="binary" precision="double">
    <group>
      <sampling>
        <fourier_space>   no    no no</fourier_space>
        <lattice>         16    1  1</lattice>
        <moments>atoms molecules gatoms gmolecules</moments>
        <![CDATA[
          atoms=phi(2)*phi(1);
          molecules=phi(4)*phi(3);
          gatoms=conj(phi(5))*phi(5);
          gmolecules=conj(phi(6))*phi(6);
        ]]>
      </sampling>
    </group>
    <group>
      <sampling>
        <fourier_space>    no   no  no</fourier_space>
        <lattice>           0    16   0</lattice>
        <moments>rn_1 rn_2 grn_1 grn_2 excitedn</moments>
        <![CDATA[
    rn_1 = phi(2)*phi(1);
    rn_2 = phi(4)*phi(3);
    grn_1 = conj(phi(5))*phi(5);
    grn_2 = conj(phi(6))*phi(6);
    excitedn = g*g/4*phi(2)*phi(2)*phi(1)*phi(1)+F*F*phi(4)*phi(3)
     - F*g/2*(phi(2)*phi(2)*phi(3)+phi(1)*phi(1)*phi(4));
        ]]>
      </sampling>
    </group>
    <group>
      <sampling>
        <fourier_space>   no no no</fourier_space>
        <lattice>         4    8  16</lattice>
        <moments>atomsr moleculesr atomsi moleculesi</moments>
        <![CDATA[
          atomsr=phi(1);
          moleculesr=phi(3);
          atomsi=-i*phi(5);
          moleculesi=-i*phi(6);
        ]]>
      </sampling>
    </group>
  </output>

</simulation>
```

This simulation is identical in function to the highdim.xmds example above, but describes the fields as an array of components rather than a list. This notation may be very valuable

when the numbers of component get very large and the equations can be easily described in terms of the index.

WARNING: There is no bounds checking on the index of your field, so be careful when writing your equations in this form.

When using this notation, if XMDS needs to calculate the k-space operator of any of the components of an array, all of them are calculated. This makes, for example, highdim_vector_version.xmds slower than the old version.

IMPORTANT: For interaction picture algorithms, if a k-space operator is applied to any component of a vector, then it is applied to ALL OF THEM. This means that highdim_vector_version.xmds only solves the correct equations when used with an EX algorithm.

# Part IV

# Reference Manual

# 11 Language Reference

## 11.1 simulation

*required* `<simulation>` **xmds tags** `</simulation>`

**Contains:** `<name>`, `<prop_dim>`, `<error_check>`, `<stochastic>`, `<globals>`, `<use_mpi>`, `<MPI_Method>`, `<field>`, `<sequence>`, `<output>`, `<noises>`, `<paths>`, `<benchmark>`, `<binary_output>` (obsolete), `<use_wisdom>`, `<use_double>` (obsolete), `<use_prefs>`, `<argv>`, `<threads>`, `<use_openmp>`, `<fftw_version>`

**Subelement of:** None

**Path to tag:** `<simulation>`

**Description:** Container tag for the **xmds** simulation code.

**Example:**

```
<simulation>
   <!-- xmds tags -->
</simulation>
```

## 11.2 name (simulation)

*optional* `<name>` **string** `</name>`

**Contains:** string

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<name>`

**Description:** The name of the **xmds** simulation. Defines the name of the output C code file and subsequently the name of the simulation binary executable. This tag is optional, and if not specified then the output filenames will be derived from the xmds script filename minus its last extension. For instance, for the `atomlaser.xmds` script, the base for the output `.cc` file will be `atomlaser`. This isn't really a very instructive example...

**Example:**

```
<simulation>
    <name> atomlaser </name>
</simulation>
```

## 11.3 prop_dim (simulation)

*required* `<prop_dim>` **string** variableName `</prop_dim>`

**Contains:** string

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<prop_dim>`

**Description:** The name of the main propagation direction. This name must appear in the equations supplied. This condition, however, is not checked and therefore a possible source of error for the user.

**Example:**

```
<simulation>
    <prop_dim> z </prop_dim>
<simulation>
```

## 11.4 error_check

*optional* `<error_check>` **bool** `</error_check>`

**Contains:** boolean

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<error_check>`

**Description:** Whether or not to run the simulation at half the time step as well as at the full time step and give the difference between the results. Defaults to `yes`.

**Example:**

```
<simulation>
    <error_check> yes </error_check>
</simulation>
```

## 11.5 use_mpi

*optional* `<use_mpi>` **bool** `<use_mpi>`

**Contains:** boolean

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<use_mpi>`

**Description:** Whether or not to use MPI routines for parallel processing of the simulation. This writes very different code depending on whether the simulation is stochastic or non-stochastic. Only certain problems on certain systems can be efficiently parallelised when the equations are non-stochastic. Defaults to `no`.

**Example:**

```
<simulation>
    <use_mpi> yes </use_mpi>
</simulation>
```

## 11.6 stochastic

*optional* `<stochastic>` **bool** `</stochastic>`

**Contains:** boolean

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<stochastic>`

**Description:** Defaults to `no`. Tells **xmds** whether or not the simulation uses Gaussian noise terms. If this tag is set to `yes` then the `<paths>`, `<seeds>` and `<noises>` tags become compulsory.

**Example:**

```
<simulation>
    <stochastic> no </stochastic>
</simulation>
```

## 11.7 MPI_Method

*optional* <MPI_Method> **string** </MPI_Method>

**Contains:** string

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <MPI_Method>

**Description:** Defaults to Scheduling. For stochastic simulations using MPI, it tells **xmds** which method to use for splitting the paths between different processors. The default "Scheduling" option is usually optimal, but requires the use of threads. If threads are unavailable, the "Uniform" option should be used.

**Example:**

```
<simulation>
    <MPI_Method> Uniform </MPI_Method>
</simulation>
```

## 11.8 paths

*optional* <paths> **int** </paths>
(*required* if <stochastic> is yes)

**Contains:** integer

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <paths>

**Description:** The number of stochastic paths to integrate.

**Example:**

```
<simulation>
    <paths> 3 </paths>
</simulation>
```

## 11.9 seed

*optional* `<seed>` **int int** `</seed>`
(*required* if `<stochastic>` is `yes`)

**Contains:** array of two integers

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<seed>`

**Description:** The seed to the random number generator. Internally, these are the seeds passed to the C routine `erand48()` to generate two independent random number generators, hence the use of two integers.

**Example:**

```
<simulation>
    <seed> 1 2 </seed>
</simulation>
```

## 11.10 noises

*optional* `<noises>` **int** `</noises>`
(*required* if `<stochastic>` is `yes`)

**Contains:** integer

**Attributes:** *optional* `kind="gaussian|gaussFast|poissonian|uniform"`, `mean="int"` (*required* for `poissonian`)

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<noises>`

**Description:** The number of noise terms in the simulation. **xmds** automatically declares the variables `n_1` ... `n_`*m* where *m* is the number of noises specified. As of **xmds**-1.3-3 the `<noises>` tag now accepts attributes. Currently these are `gaussian`, `gaussFast`, `poissonian`, and `uniform`. Each refers to a different noise source;

`gaussian` refers to the original noise routine that **xmds** uses. It produces a Gaussian-distributed variable with mean of zero and variance of 1/(product of the propagation and transverse step sizes).

`gaussFast` is a slightly faster implementation of the same routine, and will eventually replace the old `gaussian` routine.

`poissonian` is a Poissonian noise source, and requires the `mean_rate` attribute to be set to the desired mean rate of the Poissonian noise distribution. Simulations where the mean rate is a function of the propagation dimension or other variables are not supported via this method, but judicious use of the functions element can allow this rate to be adjusted manually.

`uniform` gives uniformly distributed noise on the interval $[0, 1)$.

**Example:**

```
<simulation>
    <noises kind="gaussian"> 1 </noises>
</simulation>
```

## 11.11 benchmark

*optional* <benchmark> **bool** </benchmark> xmds-1.2+

**Contains:** boolean

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <benchmark>

**Description:** Defaults to `no`. Tells **xmds** whether or not to put timing code around the main section of code (that part of the code excluding fftw creation and deletion) as a way of benchmarking the simulation, or at least giving an idea of how long the main section of code will take.

**Example:**

```
<simulation>
    <benchmark> yes </benchmark>
</simulation>
```

## 11.12 binary_output

*optional* <binary_output> **bool** </binary_output> xmds-1.2+
OBSOLETE: see <output>

**Contains:** boolean

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <binary_output>

**Description:** Defaults to `no`. If set to `yes` **xmds** saves data file in binary format instead of the default, ascii.

This tag is now obsolete. Please specify the `format` attribute in the `<output>` tag. For example: `<output format="binary">`.

**Example:**

```
<simulation>
    <binary_output> yes </binary_output>
</simulation>
```

## 11.13 use_wisdom

*optional* <use_wisdom> **bool** </use_wisdom> xmds-1.2+

**Contains:** boolean

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <use_wisdom>

**Description:** Defaults to `no`. If `yes` the simulation will use fftw's wisdom feature. The simulation will look in the user's `~/.xmds/wisdom` directory (or failing the existence of such a directory, in the directory local to the simulation binary executable) for a file labelled <hostname>.wisdom where <hostname> is the name of the computer currently running the simulation. If such a file exists, then the **xmds** simulation will load this wisdom as part of the fftw plan creation step, thus drastically reducing startup time of the simulation. Any accumulated wisdom will be then added back to this file at the fftw plan deletion stage. If no wisdom file is found, the simulation will create one in an appropriate location and then save any accumulated wisdom to this file.

**Example:**

```
<simulation>
    <use_wisdom> no </use_wisdom>
</simulation>
```

## 11.14 use_double

*optional* <use_double> **bool** </use_double> xmds-1.2+
OBSOLETE: see <output>

**Contains:** boolean

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <use_double>

**Description:** Defaults to yes. Decides the precision of the output data. Only useful when <use_binary> is set to yes. If set to no then single precision output is used. This option is useful in reducing the size of the output files.

This tag is now obsolete. Please specify the precision attribute in the <output> tag. For example: <output precision="double">.

**Example:**

```
<simulation>
    <use_double> no </use_double>
</simulation>
```

## 11.15 use_prefs

*optional* <use_prefs> **bool** </use_prefs> xmds-1.2+

**Contains:** boolean

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <use_prefs>

**Description:** Defaults to yes. Tells **xmds** whether or not to use the user preferences file (called xmds.prefs) located in either the user's .xmds directory or within the directory local to the **xmds** simulation script. By default, **xmds** will use the preferences file if it exists, if not then it uses the preferences set when **xmds** was built. One can explicitly use the build preferences by setting <use_prefs> to no.

The format of the preferences file is a sequence of key/value pairs separated by an equals sign (=). For instance, if one wished to change the compiler used by **xmds** to compile simulations then one needs to change the XMDS_CC variable. Hence, to do set this to icc, for example, one would enter the following line into the xmds.prefs file: XMDS_CC = icc

**Example:**

```
<simulation>
    <use_prefs> no </use_prefs>
</simulation>
```

## 11.16 threads

*optional* `<threads>` **int** `</threads>`

**Contains:** integer

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<threads>`

**Description:** If set to a value of $n \neq 1$, this tag tells **xmds** to create a simulation that uses the threaded version of the FFT library, and to use $n$ threads for calculating the FFT's.

**Example:**

```
<simulation>
        <threads> 3 </threads>
</simulation>
```

## 11.17 use_openmp

*optional* `<use_openmp>` **bool** `</use_openmp>`

**Contains:** boolean

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<use_openmp>`

**Description:** Defaults to `no`. Tells **xmds** whether to make use of OpenMP compiler directives to instruct the compiler how to parallelise parts of the simulation other than the FFT's. The simulation will compile correctly if OpenMP by the compiler, and you will need to ensure that the correct flags are passed to the compiler to ensure that it does enable OpenMP support (e.g. for Intel's C compiler the flag `-openmp` must be passed to the compiler to enable OpenMP). OpenMP cannot be used simultaneously with MPI for deterministic simulations. Also, the number of OpenMP threads used in a simulation defaults to the number of physical processors available. Note that best performance will be achieved if FFTW is compiled to use OpenMP threads instead of the default `pthreads` if your simulations make use of OpenMP threads.

See the OpenMP website (http://www.openmp.org) for more information about OpenMP. As of mid-2006, Intel's icc compiler supports OpenMP, however the current release of GCC (4.1.1) does not. Support for OpenMP is planned for the 4.2 release of GCC.

**Example:**

```
<simulation>
    <use_openmp> yes </use_openmp>
</simulation>
```

## 11.18 fftw_version

*optional* `<fftw_version>` **int** `</fftw_version>`

**Contains:** integer

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<fftw_version>`

**Description:** Defaults to 2. This tag tells **xmds** which version of the fftw library to use. Currently, fftw3 does not support distributed-memory fourier transforms using MPI, but only shared-memory transforms using threads. Consequently, fftw version 3 can only be used for simulations that are not MPI-using deterministic simulations. Stochastic MPI simulations can take advantage of fftw3 as they do not need distributed-memory fourier transforms.

To override the libraries that **xmds** links against when compiling a simulation that uses fftw3, override the `FFTW3_LIBS` library in your `xmds.prefs` file.

**Example:**

```
<simulation>
    <fftw_version> 3 </fftw_version>
</simulation>
```

## 11.19 globals

*optional* `<globals>` **C code** `</globals>`

**Contains:** `CDATA` block with C++ code

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<globals>`

**Description:** Defines variables and constants that are globally available to all sections of the output C code.

**Example:**

```
<simulation>
    <globals>
        <![CDATA[
            const double energy = 4;
            const double vel = 0.0;
            const double hwhm = 1.0;
        ]]>
    </globals>
<simulation>
```

## 11.20 argv

*optional* <argv> **xmds tags** </argv> xmds-1.2+

**Contains:** <arg>

**Subelement of:** <simulation>

**Path to tag:** <simulation> → <argv>

**Description:** Overall tag containing the arguments to be supplied at the simulation command line. Command line arguments can be very useful if one wants to vary parameters within the simulation, between simulation runs. Hence one can write a (Perl/Python/shell) script to run the simulation over the various parameters, without having to rewrite an **xmds** script, recompile and then rerun.

**Example:**

```
<simulation>
    <argv>
        <!-- xmds tags -->
    </argv>
<simulation>
```

### 11.20.1 arg

*required* <arg> **xmds tags** </arg> xmds-1.2+

**Contains:** <name>, <type>, <default_value>

**Subelement of:** <argv>

**Path to tag:** <simulation> → <argv> → <arg>

**Description:** Container for the tags describing a given command line argument.

**Example:**

```
<simulation>
    <argv>
        <arg>
            <!-- xmds tags -->
        </arg>
    </argv>
<simulation>
```

### 11.20.2 name (arg)

*required* <name> **char** * </name> xmds-1.2+

**Contains:** string

**Subelement of:** <arg>

**Path to tag:** <simulation> → <argv> → <arg> → <name>

**Description:** The name of the command line argument. This tag is used to create a flag to specify the value of the variable entered at the command line. Two forms are accepted: a short form and a long form, each conforming to the GNU getopt conventions. For instance, for a variable named `nconst` the flag used at the command line will have a short form of `-n` and a long form of `--nconst`. However, if another variable has been chosen beginning with the letter n, then `-n` is no longer unique, and to make the short flag unique, **xmds** chooses the next character in the variable name, in this case the letter c, and making the short form flag `-c`. The long form remains the same, i.e. `--nconst`. Of course, if the letter c has also been used then **xmds** loops through all of the other letters in the variable name until it finds a match. If no match is found **xmds** reports an error.

To check the short and long forms of the flags the simulation expects, the user can use the `-h` or `--help` options with the simulation. For instance, with the `nlse` simulation, one can enter the following command to get a listing of the arguments the **xmds** simulation expects at its command line:

```
% nlse --help
```

**Example:**

```
<simulation>
    <argv>
        <arg>
            <name> nconst </name>
            <type> int </type>
            <default_value> 2 </default_value>
```

```
        </arg>
    </argv>
<simulation>
```

### 11.20.3 type (arg)

*required* <type> **char *** </type> xmds-1.2+

**Contains:** string

**Subelement of:** <arg>

**Path to tag:** <simulation> → <argv> → <arg> → <type>

**Description:** The type of the command line argument, e.g. int, double, char *. At present this mechanism cannot handle complex inputs.

**Example:**

```
<simulation>
    <argv>
        <arg>
            <name> nconst </name>
            <type> int </type>
            <default_value> 2 </default_value>
        </arg>
    </argv>
<simulation>
```

### 11.20.4 default_value

*required* <default_value> **char *** </default_value> xmds-1.2+

**Contains:** string

**Subelement of:** <arg>

**Path to tag:** <simulation> → <argv> → <arg> → <default_value>

**Description:** The default value of the command line argument. Values will be converted from the string entered into the type given in the <type> tag. If the variable is not specified at the command line then this is the value used by the simulation.

**Example:**

```
<simulation>
    <argv>
        <arg>
            <name> nconst </name>
            <type> int </type>
            <default_value> 2 </default_value>
        </arg>
    </argv>
<simulation>
```

## 11.21 field

*required* `<field>` **xmds tags** `</field>`

**Contains:** `<name>`, `<dimensions>`, `<lattice>`, `<domains>`, `<samples>`, `<vector>`

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<globals>` → `<field>`

**Description:** Container element to hold the tags describing the field to be integrated. At present, only one field is permitted in a simulation.

**Example:**

```
<simulation>
    <field>
        <!-- xmds tags -->
    </field>
<simulation>
```

### 11.21.1 name (field)

*optional* `<name>` **string** `</name>`

**Contains:** string

**Subelement of:** `<field>`

**Path to tag:** `<simulation>` → `<field>` → `<name>`

**Description:** The name of the field to integrate. Defaults to `main`.

**Example:**

```
<simulation>
    <field>
        <name> main </name>
    </field>
</simulation>
```

### 11.21.2 dimensions

*optional* <dimensions> **string** variableName **string** variableName . . . </dimensions>

**Contains:** array of strings

**Subelement of:** <field>

**Path to tag:** <simulation> → <field> → <dimensions>

**Description:** A space delimited array of the names of dimensions in the field other than the propagation dimension. This element is therefore a list of the transverse field dimensions.

**Example:**

```
<simulation>
    <field>
        <dimensions> t </dimensions>
    </field>
<simulation>
```

### 11.21.3 lattice (field)

*optional* <lattice> **int int** . . . </lattice>
(*required* if <dimensions> assignment present)

**Contains:** array of integers

**Subelement of:** <field>

**Path to tag:** <simulation> → <field> → <lattice>

**Description:** A space delimited array of integers giving the number of points in the grid for each dimension listed in the <dimensions> tag.

**Exmaple:**

```
<simulation>
    <field>
        <lattice> 100 </lattice>
    </field>
</simulation>
```

### 11.21.4 domains

*optional* `<domains>` **(double, double) (double, double)** ... `</domains>`
(*required* if `<dimensions>` assignment present)

**Contains:** array of ordered pairs of doubles

**Subelement of:** `<field>`

**Path to tag:** `<simulation>` → `<field>` → `<domains>`

**Description:** This tag specifies the domain range of each dimension listed in the `<dimensions>` tag.

**Example:**

```
<simulation>
    <field>
        <domains> (-5, 5) </domains>
    </field>
</simulation>
```

### 11.21.5 samples (field)

*required* `<samples>` **int int** ... `</samples>`

**Contains:** array of integers, specifically either `1` or `0`

**Subelement of:** `<field>`

**Path to tag:** `<simulation>` → `<field>` → `<samples>`

**Description:** Tells **xmds** which moment groups to sample. This tag should contain a space separated list of `1`'s and `0`'s. A `1` meaning sample the moment group in question, and a `0` meaning do not.

**Example:**

```
<simulation>
    <field>
        <samples> 1 </samples>
    </field>
</simulation>
```

### 11.21.6 vector

*required* <vector> **xmds tags** </vector>

**Contains:** <name>, <filename>, <type>, <components>, <fourier_space>, <vectors>, CDATA

**Subelement of:** <field>

**Path to tag:** <simulation> → <field> → <vector>

**Description:** A container for tags describing a vector of the field.

**Example:**

```
<simulation>
    <field>
        <vector>
            <!-- xmds tags -->
        </vector>
    </field>
</simulation>
```

#### 11.21.6.1 name (vector)

*required* <name> **string** </name>

**Contains:** string

**Subelement of:** <vector>

**Path to tag:** <simulation> → <field> → <vector> → <name>

**Description:** Name of the vector in question. At least one vector must be present, and at least one vector must be called `main`.

**Example:**

```
<simulation>
    <field>
        <vector>
            <name> main </name>
        </vector>
    </field>
</simulation>
```

#### 11.21.6.2 filename (vector)

*optional* `<filename>` **string** `</filename>`

**Contains:** string

**Attributes:** *optional* `format="ascii"|"binary"|"xsil"`

**Attributes for the XSIL format:** *optional* `moment_group="N"`
*optional* `geometry_matching_mode="strict"|"loose"`

**Subelement of:** `<vector>`

**Path to tag:** `<simulation>` → `<field>` → `<vector>` → `<filename>`

**Description:** Tells **xmds** the file from which to load the initial field.

The `<filename>` tag accepts one optional attribute: `format`. This attribute can take one of three options; `"ascii"`, `"binary"`, or `"xsil"` where `"ascii"` is the default option. However, `"xsil"` is the most robust of these formats as it can load any binary XSIL file produced by XMDS (irrespective of what architecture the file was produced on). See Chapter 2, Section 2.6.1 for more information on loading XSIL files.

Caution should be taken here when using the `"binary"` format since loading a binary file is more difficult to get correct than loading an ascii file. However, the added difficulty is offset by being able to have smaller and more complex input data. A first thing to note is that the byte ordering of the system you are going to be loading the file into has to be the same as the file itself. For instance, creating a binary file for input to **xmds** on PowerPC **will not** load correctly on an x86 platform.

For the `"binary"` and `"ascii"` formats, **xmds** expects the input data to be essentially interlaced so that it is loaded into memory correctly. The best way to explain this is by way of example. If the input data is to be three vectors, say `x`, `y` and `z`, then **xmds** expects the data to be formed thus:

```
x[0] y[0] z[0] x[1] y[1] z[1] ...
```

and so on with a new line or space between each entry. The only difference between the ascii and binary input formats is that the newline (or space) character is unnecessary, and **xmds** just expects a sequence of **double** values ordered as above.

For complex types, **xmds** expects the imaginary part of each variable to follow the real part. If the input data for the previous example were complex numbers, then **xmds** would expect the data to be in the order:

```
real(x[0]) imag(x[0]) real(y[0]) imag(y[0]) real(z[0]) imag(z[0]) ...
```

**Example:**

```
<simulation>
    <field>
        <vector>
            <filename format="binary"> blah </filename>
        </vector>
    </field>
</simulation>
```

#### 11.21.6.3 type (vector)

*optional* <type> **string** of complex or double </type>

**Contains:** string interpreted as either **complex** or **double** type

**Subelement of:** <vector>

**Path to tag:** <simulation> → <field> → <vector> → <type>

**Description:** The data type of the vector. Defaults to **complex** if not specified. It is a good idea to use **complex** here when one is using a Fourier transform technique to integrate the equations, even if initially the variables are **double**.

**Example:**

```
<simulation>
    <field>
        <vector>
            <type> complex </type>
        </vector>
    </field>
</simulation>
```

#### 11.21.6.4 components

*required* <components> **string** variableName **string** variableName ... </components>

**Contains:** array of strings

**Subelement of:** <vector>

**Path to tag:** `<simulation>` → `<field>` → `<vector>` → `<components>`

**Description:** Names the components of the vector. If a large array of components are required, it is possible to declare that here by putting the size of the array in parentheses immediately after the name. Subsequent references to these components must have the index afterwards in parentheses.

**Example:**

```
<simulation>
    <field>
        <vector>
            <components> phi JoeIsGreat(256) </components>
        </vector>
    </field>
</simulation>
```

#### 11.21.6.5 fourier_space (vector)

*optional* `<fourier_space>` **bool bool** ... `</fourier_space>`
(*required* if `<filename>` not present)

**Contains:** array of booleans

**Subelement of:** `<vector>`

**Path to tag:** `<simulation>` → `<field>` → `<vector>` → `<fourier_space>`

**Description:** Tells **xmds** whether the dimension specified is initialised in Fourier space. This is a space separated list of `yes`/`no` values. A value of `yes` meaning the dimension is defined in Fourier space, and `no` meaning the dimension is defined in $x$-space.

**Example:**

```
<simulation>
    <field>
        <vector>
            <fourier_space> no </fourier_space>
        </vector>
    </field>
</simulation>
```

#### 11.21.6.6 vectors (vector)

*required* `<vectors>` **string** variableName **string** variableName ... `</vectors>`

**Contains:** array of strings

**Subelement of:** `<vector>`

**Path to tag:** `<simulation>` → `<field>` → `<vector>` → `<vectors>`

**Description:** Tells **xmds** the names of the variables are to be referenced in a `CDATA` block. Defaults to `main`.

**Example:**

```
<simulation>
    <field>
        <vector>
            <vectors> main vc1 </vectors>
        </vector>
    </field>
</simulation>
```

## 11.22 sequence

*required* `<sequence>` **xmds tags** `</sequence>`

**Contains:** `<integrate>`, `<filter>`, `<sequence>`

**Subelement of:** `<simulation>` or `<sequence>`

**Path to tag:** `<simulation>` → `<sequence>` (→ `<sequence>`)

**Description:** Container tag for the sequence of integrations to perform. May contain other sequences within itself. If subsequences exist, then they must contain a `<cycles>` assignment.

**Example:**

```
<simulation>
    <sequence>
        <!-- xmds tags -->
    </sequence>
</simulation>
```

### 11.22.1 cycles

*optional* `<cycles>` **int** `</cycles>`
(*required* in nested `<sequence>`s)

**Contains:** integer

**Subelement of:** <sequence>

**Path to tag:** <simulation> → <sequence> → <cycles>

**Description:** The number of times to perform a given sequence (as a subsequence of the main sequence). Defaults to 1.

**Example:**

```
<simulation>
    <sequence>
        <cycles> 3 </cycles>
    </sequence>
</simulation>
```

### 11.22.2 integrate

*optional* <integrate> **xmds tags** </integrate>

**Contains:** <algorithm>, <interval>, <iterations>, <tolerance>, <max_iterations>, <smallmemory>, <cutoff>, <halt_non_finite>, <lattice>, <samples>, <k_operators>, <moment_group>, <functions>, <vectors>, CDATA

**Subelement of:** <sequence>

**Path to tag:** <simulation> → <sequence> → <integrate>

**Description:** Container element holding the tags that describe how the integration should take place.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <!-- xmds tags -->
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.1 algorithm

*optional* <algorithm> **string** algorithmName </algorithm>

**Contains:** string, one of RK4EX, RK4IP, ARK45EX, ARK45IP, SIEX, or SIIP

**Subelement of:** <integrate>

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<algorithm>`

**Description:** The algorithm to use when integrating the equations. Defaults to `SIEX` if `<stochastic>` is `yes` or `RK4EX` if `<stochastic>` is `no`. The six algorithms that **xmds** currently contains are:

`RK4EX` Fourth-order Runge-Kutta in the explicit picture.

`RK4IP` Fourth-order Runge-Kutta in the interaction picture.

`ARK45EX` adaptive step size Runge-Kutta-Fehlberg in the explicit picture.

`ARK45IP` adaptive step size Runge-Kutta-Fehlberg in the interaction picture.

`SIEX` Semi-implicit method in the explicit picture.

`SIIP` Semi-implicit method in the interaction picture.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <algorithm> RK4IP </algorithm>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.2 interval

*required* `<interval>` **double** `</interval>`

**Contains:** double

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<interval>`

**Description:** The integration range. This is the interval over which the main propagation dimension will be integrated.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <interval> 20 </interval>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.3 iterations

*optional* `<iterations>` **int** `</iterations>`

**Contains:** integer

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<iterations>`

**Description:** When using one of the semi-implicit algorithms this option can be altered to control the number of iterations of the algorithm to for convergence of the method. Defaults to 3.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <iterations> 5 </iterations>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.4 tolerance

*required* `<tolerance>` **double** `</tolerance>`

**Contains:** double

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<tolerance>`

**Description:** If one of the adaptive step size methods is used, this assignment controls the maximum *relative* error that is allowed per step on any of the grid points.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <tolerance>1e-10</tolerance>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.5 max_iterations

*optional* `<max_iterations>` **int** `</max_iterations>`

**Contains:** integer

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<max_iterations>`

**Description:** Applies to the ARK45 algorithms only. If the behaviour of the solution is unknown and might result in a very small step size the maximum number of iterations (steps and discarded steps) can be limited with this optional tag. This is particularly useful in the debugging stage or on systems that impose time limits on their running programs.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <max_iterations>5000000</max_iterations>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.6 min_time_step

*optional* `<min_time_step>` **double** `</min_time_step>`

**Contains:** double

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<min_time_step>`

**Description:** The minimum time-step an integration algorithm should try, before halting that pass of the integration. This option applies to the adaptive algorithms (ARK45 and ARK89) only.

The default value is `1e-13`, that is, $10^{-13}$. A value of `0` disables the check on the time-step, which will slightly speed up integrations that do not require the check.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <min_time_step>1e-10</min_time_step>
```

```
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.7 smallmemory

*optional* <smallmemory> **bool** </smallmemory>

**Contains:** boolean

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <sequence> → <integrate> → <smallmemory>

**Description:** Defaults to no. In this case, when using the ARK45IP algorithm **xmds** calculates and stores six copies of the derivative operators per step as each of them can be reused once. For problems with memory limitations it might be better to assign yes to this element, then each array of derivatives needs to be calculated twice per step, but significantly less memory should be used.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <smallmemory> yes </smallmemory>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.8 cutoff

*optional* <cutoff> **double** </cutoff>

**Contains:** double

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <sequence> → <integrate> → <cutoff>

**Description:** If one of the adaptive step size methods is used, the cutoff value sets the threshold for the function values that are included in the determination of relative errors. Grid points where the function is less than cutoff*peakvalue are not included. The value defaults to 1/1000;

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <cutoff> 1e-5 </cutoff>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.9 halt_non_finite

*optional* `<halt_non_finite>` **bool** `</halt_non_finite>`

**Contains:** boolean

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<halt_non_finite>`

**Description:** If `yes`, then halt the current integration pass if it produces a non-finite number in the first component of the main vector. Since non-finite numbers usually propogate quickly through the components of the main vector, this will quickly detect any non-finite number.

Examples of non-finite numbers are `1.0/0.0`, which is infinite, and `0.0/0.0`, which is not a number (NaN). Some platforms process non-finite numbers slower than finite numbers; others process them faster. Regardless, they are unlikely to be useful results from an integration.

Note that, by default, **xmds** compiles the generated C++ code with optimizations that sacrifice strict compliance with floating-point arithmetic standards (e.g. IEEE 754).

Many of these optimizations assume that all floating-point numbers are finite. Naturally, this may pose problems for a non-finite number check. In some cases, the check for non-finite numbers is optimized to oblivion. Depending on other optimizations, the time-step may become NaN and the integration will never move forward and therefore never terminate.

An easy way to see if the check for non-finite numbers survived optimization is to run the following (supposing **xmds** compiled the `simulation` binary from `simulation.xmds`):

```
strings simulation | grep halt_non_finite:
```

(The colon is important.) If this command prints output similar to the following:

```
* NOTICE: halt_non_finite: Integration halted.
```

then at least the check is not optimized away and `halt_non_finite` is more likely to work.

When using GCC **xmds** enables the `-ffast-math` flag. In GCC version 3.3, this seems to pose problems with `halt_non_finite`. Removing the `-ffast-math` flag should solve the problem. Alternatively, keeping this flag and adding the `-fno-unsafe-math-optimizations` flag *after* the `-ffast-math` flag seems to work (and should produce faster code). GCC 4 seems to have no problem.

There seems to be no problem using version 10.1 of Intel's compiler `icc`. Note that `icc` sacrifices floating-point compliance for speed by default, but this can be changed using the `-fp-model` flag.

Remember you can change compilation flags by editing the preference file (see section 2.8 for more information on preferences). Of course, you can also edit the compilation command printed by **xmds** and run it by hand.

Overall, however, there are no guarantees that `halt_non_finite` will work when sacrificing accuracy for speed in floating-point arithmetic.

Defaults to `no`.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <halt_non_finite> yes </halt_non_finite>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.10 lattice (integrate)

*required* `<lattice>` **int** `</lattice>`

**Contains:** integer

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<lattice>`

**Description:** The number of points to use over the integration interval i.e. over the number entered in the `<interval>` assignment.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <lattice> 1000 </lattice>
        </integrate>
```

```
    </sequence>
</simulation>
```

#### 11.22.2.11 samples (integrate)

*required* <samples> **int int** ... </samples>

**Contains:** array of integers

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <sequence> → <integrate> → <samples>

**Description:** The number of samples to take of each output moment group. This is a space separated list of integers, the number of which must be equal to the number of output moment groups. Each integer must be a factor of the <lattice> assignment or 0. If set to 0 then the given moment group is not sampled.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <samples> 50 </samples>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.12 k_operators

*optional* <k_operators> **xmds tags** </k_operators>

**Contains:** <vectors>, <constant>, <operator_names>, CDATA

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <sequence> → <integrate> → <k_operators>

**Description:** Container for tags describing the $k$-space (i.e. Fourier space) operators.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <k_operators>
                <!-- xmds tags -->
            </k_operators>
```

```
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.12.1 vectors (k_operators)

*optional* <vectors> **string** variableName **string** variableName ... </vectors>

**Contains:** array of strings

**Subelement of:** <k_operators>

**Path to tag:** <simulation> → <sequence> → <integrate> → <k_operators> → <vectors>

**Description:** Vectors to be referred to in CDATA block. Defaults to main.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <k_operators>
                <vectors> main vc1 </vectors>
            </k_operators>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.12.2 constant

*optional* <constant> **bool** </constant>

**Contains:** boolean

**Subelement of:** <k_operators>

**Path to tag:** <simulation> → <sequence> → <integrate> → <k_operators> → <constant>

**Description:** Tells **xmds** whether or not the $k$-space operators are constant over the course of the simulation, in other words, they don't depend upon the propagation dimension. If they are constant, then giving a value of yes for this tag speeds up the simulation as more efficient code can be used. Defaults to no.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <k_operators>
                <constant> yes </constant>
            </k_operators>
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.12.3 operator_names

*required* <operator_names> **string** variableName **string** variableName ... </operator_names>

**Contains:** array of strings

**Subelement of:** <k_operators>

**Path to tag:** <simulation> → <sequence> → <integrate> → <k_operators> → <operator_names>

**Description:** The names of the $k$-space operators as they appear in the CDATA block within the <k_operators> element. This is a space separated list of strings of the operator names.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <k_operators>
                <operator_names> L </operator_names>
            </k_operators>
        </integrate>
    </sequence>
</simulation>
```

### 11.22.2.13 moment_group (integrate)

*optional* <moment_group> **xmds tags** </moment_group>

**Contains:** <moments>, <integrate_dimension>, CDATA

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <sequence> → <integrate> → <moment_group>

**Description:** Defines and calculates a number or vector integrated through zero or more of the transverse dimensions of the problem.

**Example:**

```
<simulation>
  <sequence>
      <integrate>
         <moment_group>
            <moments>joe is great</moments>
            <integrate_dimension>no yes</integrate_dimension>
          <![CDATA[
                joe += ~psi*psi;
                is += ~phi*phi;
                great += theta;
                ]]>
        </moment_group>
      </integrate>
   </sequence>
</simulation>
```

#### 11.22.2.14 functions (integrate)

*optional* `<functions>` `CDATA` `</functions>`

**Contains:** `CDATA`

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<functions>`

**Description:** This is the best place to define any functions that do not depend on the transverse dimensions.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <functions>
             <![CDATA[
                /* Some C code */
               ]]>
            </functions >
        </integrate>
    </sequence>
</simulation>
```

#### 11.22.2.15 vectors (integrate)

*required* `<vectors>` **string** variableName **string** variableName ... `</vectors>`

**Contains:** array of strings

**Subelement of:** `<integrate>`

**Path to tag:** `<simulation>` → `<sequence>` → `<integrate>` → `<vectors>`

**Description:** The vectors **xmds** needs to access in the equations. Given as a space separated list of strings. Defaults to `main`.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <vectors> main vc1 </vectors>
        </integrate>
    </sequence>
</simulation>
```

### 11.22.3 filter

*optional* `<filter>` **xmds tags** `</filter>`

**Contains:** `<vectors>`, `<fourier_space>`, `CDATA`

**Subelement of:** `<sequence>`

**Path to tag:** `<simulation>` → `<sequence>` → `<filter>`

**Description:** Container element for the tags describing how the field is to be filtered, if at all.

**Example:**

```
<simulation>
    <sequence>
        <filter>
            <!-- xmds tags -->
        </filter>
    </sequence>
</simulation>
```

#### 11.22.3.1 moment_group (filter)

*optional* `<moment_group>` **xmds tags** `</moment_group>`

**Contains:** `<moments>`, `<integrate_dimension>`, `CDATA`

**Subelement of:** `<filter>`

**Path to tag:** `<simulation>` → `<sequence>` → `<filter>` → `<moment_group>`

**Description:** Defines and calculates a number or vector integrated through zero or more of the transverse dimensions of the problem.

**Example:**

```
<simulation>
  <sequence>
      <filter>
         <moment_group>
            <moments>joe is great</moments>
            <integrate_dimension>no yes</integrate_dimension>
          <![CDATA[
                joe += ~psi*psi;
                is += ~phi*phi;
                great += theta;
                ]]>
        </moment_group>
      </filter>
   </sequence>
</simulation>
```

#### 11.22.3.2 functions (filter)

*optional* `<functions>` `CDATA` `</functions>`

**Contains:** `CDATA`

**Subelement of:** `<filter>`

**Path to tag:** `<simulation>` → `<sequence>` → `<filter>` → `<functions>`

**Description:** This is the best place to define any functions that do not depend on the transverse dimensions.

**Example:**

```
<simulation>
    <sequence>
        <filter>
            <functions>
             <![CDATA[
                /* Some C code */
              ]]>
            </functions >
        </filter>
    </sequence>
</simulation>
```

#### 11.22.3.3 vectors (filter)

*required* `<vectors>` **string** variableName **string** variableName ... `</vectors>`

**Contains:** array of strings

**Subelement of:** `<filter>`

**Path to tag:** `<simulation>` → `<filter>` → `<vectors>`

**Description:** The names of the vectors **xmds** should apply the filter to. Given as a space separated list of strings. Defaults to `main`.

**Example:**

```
<simulation>
        <filter>
            <vectors> main vc1 </vectors>
        </filter>
</simulation>
```

#### 11.22.3.4 fourier_space (filter)

*optional* `<fourier_space>` **boolean boolean** ... `</fourier_space>`

**Contains:** array of booleans

**Subelement of:** `<filter>`

**Path to tag:** `<simulation>` → `<integrate>` → `<filter>` → `<fourier_space>`

**Description:** Tells **xmds** in which space the filter is to be applied. This is a list of `yes`/`no` options for each vector.

**Example:**

```
<simulation>
    <integrate>
        <filter>
            <fourier_space> yes no </fourier_space>
        </filter>
    </integrate>
</simulation>
```

#### 11.22.3.5 cross_propagation

*optional* <cross_propagation> **xmds tags** </cross_propagation>

**Contains:** <vectors>, <prop_dim>, CDATA

**Subelement of:** <integrate>

**Path to tag:** <simulation> → <integrate> → <cross_propagation>

**Description:** Container of the tags describing the cross propagation vectors, if any.

**Example:**

```
<simulation>
    <integrate>
        <cross_propagation>
            <!-- xmds tags -->
        </cross_propagation>
    </integrate>
</simulation>
```

##### 11.22.3.5.1 prop_dim (cross_propagation)

*required* <prop_dim> **string** </prop_dim>

**Contains:** string

**Subelement of:** <cross_propagation>

**Path to tag:** <simulation> → <integrate> → <cross_propagation> → <prop_dim>

**Description:** The propagation dimension of the cross propagating vector.

**Example:**

```
<simulation>
    <integrate>
        <cross_propagation>
            <prop_dim> z </prop_dim>
```

```
        </cross_propagation>
    </integrate>
</simulation>
```

#### 11.22.3.5.2 vectors (cross_propagation)

*required* <vectors> **string** variableName **string** variableName ... </vectors>

**Contains:** array of strings

**Subelement of:** <cross_propagation>

**Path to tag:** <simulation> → <sequence> → <integrate> → <cross_propagation> → <vectors>

**Description:** The names of the cross propagating vectors as a list of strings.

**Example:**

```
<simulation>
    <sequence>
        <integrate>
            <cross_propagation>
                <vectors> main vc1 </vectors>
            </cross_propagation>
        </integrate>
    </sequence>
</simulation>
```

### 11.22.4 breakpoint

*optional* <breakpoint> **xmds tags** </breakpoint>

**Contains:** <filename>, <fourier_space>, <vectors>

**Subelement of:** <sequence>

**Path to tag:** <simulation> → <sequence> → <breakpoint>

**Description:** Saves some vectors to a binary XSIL file when this element is reached in the simulation. This can be used for generating an XSIL file that can be used as an input for another simulation, or to check the behaviour of the simulation part way through. This feature is described further in Chapter 2, Section 2.9.

**Example:**

```
<simulation>
    <sequence>
        <breakpoint>
            <!-- xmds tags -->
        </breakpoint>
    </sequence>
</simulation>
```

#### 11.22.4.1 filename (breakpoint)

*optional* `<filename>` **string** `</filename>`

**Contains:** string

**Subelement of:** `<breakpoint>`

**Path to tag:** `<simulation>` → `<sequence>` → `<breakpoint>` → `<filename>`

**Description:** Sets the XSIL file to which the breakpoint information should be saved.

**Example:**

```
<simulation>
    <sequence>
        <breakpoint>
            <filename> blah.xsil </filename>
        </breakpoint>
    </sequence>
</simulation>
```

#### 11.22.4.2 fourier_space (breakpoint)

*optional* `<fourier_space>` **boolean boolean** ... `</fourier_space>`

**Contains:** array of booleans

**Subelement of:** `<breakpoint>`

**Path to tag:** `<simulation>` → `<sequence>` → `<breakpoint>` → `<fourier_space>`

**Description:** Tells **xmds** in which space the breakpoint is to be written in. This is a list of `yes`/`no` options for each vector.

**Example:**

```
<simulation>
    <sequence>
        <breakpoint>
            <fourier_space> yes no </fourier_space>
        </breakpoint>
    </sequence>
</simulation>
```

#### 11.22.4.3 vectors (breakpoint)

*required* `<vectors>` **string** vectorName **string** vectorName ... `</vectors>`

**Contains:** array of strings

**Subelement of:** `<breakpoint>`

**Path to tag:** `<simulation>` → `<sequence>` → `<breakpoint>` → `<vectors>`

**Description:** The names of the vectors that will be saved to the XSIL file.

**Example:**

```
<simulation>
    <sequence>
        <breakpoint>
            <vectors> main vc1 </vectors>
        </breakpoint>
    </sequence>
</simulation>
```

## 11.23 output

*required* `<output>` **xmds tags** `</output>`

**Contains:** `<filename>`, `<group>`

**Attributes:** *optional* `format="ascii|binary"`, *optional* `precision="double|single"`

**Subelement of:** `<simulation>`

**Path to tag:** `<simulation>` → `<output>`

**Description:** Container for elements describing what data should be output and how it should be output.

Accepts two optional attributes, `format` and `precision`. The `format` attribute defines the output format of the data. The options available are `"ascii"` and `"binary"`, with `"ascii"` being the default option. The `precision` attribute defines the output data

precision. This can be either `"double"` or `"single"`, with these options referring to double or single precision floating point numbers respectively. This option is only meaningful when `format` is set to `"binary"` as the precision does not affect the output when `"ascii"` is chosen.

**Example:**

```
<simulation>
    <output format="binary" precision="single">
        <!-- xmds tags -->
    </output>
</simulation>
```

### 11.23.1 filename (output)

*optional* `<filename>` **string** `</filename>`

**Contains:** string

**Subelement of:** `<output>`

**Path to tag:** `<simulation>` → `<output>` → `<filename>`

**Description:** Optional filename for output data.

**Example:**

```
<simulation>
    <output>
        <filename> nlse.xsil </filename>
    </output>
</simulation>
```

### 11.23.2 group

*required* `<group>` **xmds tags** `</group>`

**Contains:** `<sampling>`, `<post_propagation>`

**Subelement of:** `<output>`

**Path to tag:** `<simulation>` → `<output>` → `<group>`

**Description:** Container for tags describing the relevant moment group. There must be at least one group element given.

**Example:**

```
<simulation>
    <output>
        <group>
            <!-- xmds tags -->
        </group>
    </output>
</simulation>
```

#### 11.23.2.1 sampling

*required* <sampling> **xmds tags** </sampling>

**Contains:** <vectors>, <fourier_space>, <lattice>, <moments>, CDATA

**Subelement of:** <group>

**Path to tag:** <simulation> → <output> → <group> → <sampling>

**Description:** Container for tags describing how to sample the moment group.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <!-- xmds tags -->
            </sampling>
        </group>
    </output>
</simulation>
```

##### 11.23.2.1.1 type (sampling)

*optional* <type> **string** of complex or double </type>

**Contains:** string interpreted as either **complex** or **double** type

**Subelement of:** <sampling>

**Path to tag:** <simulation> → <output> → <group> → <sampling> → <type>

**Description:** The data type of the output data. Defaults to **complex** if the vector that the output data depends on is of type **complex**, and **double** otherwise. Note that a type of **double** cannot be used with a <post_propagation> tag as the fourier transforms available to the <post_propagation> tag require the output data to be of type **complex**.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <type> complex </type>
            </sampling>
        </group>
    </output>
</simulation>
```

#### 11.23.2.1.2 fourier_space (sampling)

*required* <fourier_space> **bool bool** ... </fourier_space>

**Contains:** array of booleans

**Subelement of:** <sampling>

**Path to tag:** <simulation> → <output> → <group> → <sampling> → <fourier_space>

**Description:** A boolean telling **xmds** whether or not to sample in Fourier space. This should be a space separated list of booleans, one for each transverse dimension.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <fourier_space> no </fourier_space>
            </sampling>
        </group>
    </output>
</simulation>
```

#### 11.23.2.1.3 vectors (sampling)

*optional* <vectors> **string** variableName **string** variableName ... </vectors>

**Contains:** array of strings

**Subelement of:** <sampling>

**Path to tag:** <simulation> → <output> → <group> → <sampling> → <vectors>

**Description:** Space separated list of strings giving the names of the vectors to sample. Defaults to `main`.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <vectors> main vc1 </vectors>
            </sampling>
        </group>
    </output>
</simulation>
```

#### 11.23.2.1.4 lattice (sampling)

*optional* `<lattice>` **int int** ... `</lattice>`

**Contains:** array of integers

**Subelement of:** `<sampling>`

**Path to tag:** `<simulation>` → `<output>` → `<group>` → `<sampling>` → `<lattice>`

**Description:** A space separated list of integers, each describing how many points to sample of each transverse dimension (if greater than 1: see below). One entry must exist for each transverse dimension.

If an entry is set to `0`, then **xmds** integrates the moments over this dimension. this will cause the output field to no longer be a function of this transverse dimension.

If an entry is set to `1`, then **xmds** will sample the moments on a cross-sectional slice of this dimension, also causing the output field to lose this transverse dimension. If this dimension is in normal space then **xmds** will extract the slice at the middle lattice point (point number $N/2 + 1$ using integer division), otherwise **xmds** will extract the slice at the zero momentum point, $k = 0$.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <lattice> 50 </lattice>
            </sampling>
        </group>
    </output>
</simulation>
```

#### 11.23.2.1.5 moments (sampling)

*required* <moments> **string** variableName **string** variableName ... </moments>

**Contains:** array of strings

**Subelement of:** <sampling>

**Path to tag:** <simulation> → <output> → <group> → <sampling> → <moments>

**Description:** A list of strings of the names of the moments to sample. These variables will then be mentioned in the CDATA block following this tag.

**Example:**

```
<simulation>
    <output>
        <group>
            <sampling>
                <moments> pow_dens </moments>
            </sampling>
        </group>
    </output>
</simulation>
```

### 11.23.2.2 post_propagation

*optional* <post_propagation> **xmds tags** </post_propagation>

**Contains:** <fourier_space>, <moments>, CDATA

**Subelement of:** <group>

**Path to tag:** <simulation> → <output> → <group> → <post_propagation>

**Description:** Container for tags describing any post propagation processing of the data that should be done prior to output to file.

**Example:**

```
<simulation>
    <output>
        <group>
            <post_propagation>
                <!-- xmds tags -->
            </post_propagation>
        </group>
    </output>
</simulation>
```

#### 11.23.2.2.1 fourier_space (post_propagation)

*required* <fourier_space> **bool bool** ... </fourier_space>

**Contains:** array of booleans

**Subelement of:** <post_propagation>

**Path to tag:** <simulation> → <output> → <group> → <post_propagation> → <fourier_space>

**Description:** Whether or not the post propagation is performed in Fourier space. This is a list of yes/no entries for the propagation dimension and as many *remaining* transverse dimensions.

**Example:**

```
<simulation>
    <output>
        <group>
            <post_propagation>
                <fourier_space> no </fourier_space>
            </post_propagation>
        </group>
    </output>
</simulation>
```

#### 11.23.2.2.2 moments (post_propagation)

*required* <moments> **string** variableName **string** variableName ... </moments>

**Contains:** array of strings

**Subelement of:** <post_propagation>

**Path to tag:** <simulation> → <output> → <group> → <post_propagation> → <moments>

**Description:** The names of the moments (with different names to the moments defined directly within the group element) to be derived from the post processing.

**Example:**

```
<simulation>
    <output>
        <group>
            <post_propagation>
                <moments> pow_dens </moments>
            </post_propagation>
        </group>
```

```
    </output>
</simulation>
```

# Part V

# Appendix

# A

# The XSIL input/output syntax

Following is an example of a multi-dimensional multi-component field written in XSIL format (for more information see reference [2]). The XSIL format is extensible, and the format below has been specifically tailored for representing this type of field. The parent element, known as an `<XSIL>` data container, contains one `<Param>` assignment, and two `<Array>` assignments.

The one and only `<Param>` assignment, "`n_independent`", is an integer and defines the number of independent dimensions of the field.

The first `<Array>` element, "`variables`", defines the total number and names of all variables—both independent and dependent. The second `<Array>` element, "`data`", defines the lattice for the data, and then the data itself. Following standard C index convention, the last dimension index is the one that changes most rapidly in the data. Hence this data has 3 lattice points in the first independent dimension "`t`", 5 lattice points in the second independent dimension "`x`", and there are 4 variables to be read—2 independent and 2 dependent, which of course must be the same as the number of variable names declared in the variables array.

```
<?xml version="1.0">

<XSIL Name="moment_group_1">
  <Param Name="n_independent">2</Param>
  <Array Name="variables" Type="Text">
    <Dim>4</Dim>
    <Stream><Metalink Format="Text" Delimiter=" \n"/>
      t x real(a1) imag(a1)
    </Stream>
  </Array>
  <Array Name="data" Type="double">
    <Dim>3</Dim>
    <Dim>5</Dim>
```

```
    <Dim>4</Dim>
    <Stream><Metalink Format="Text" Delimiter=" \n"/>
      -1.000000e+00 -1.000000e+00 0.000000e+00 0.000000e+00
      -1.000000e+00 -0.500000e+00 0.000000e+00 0.000000e+00
      -1.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
      -1.000000e+00 0.500000e+00 0.000000e+00 0.000000e+00
      -1.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00
      0.000000e+00 -1.000000e+00 0.000000e+00 0.000000e+00
      0.000000e+00 -0.500000e+00 1.000000e+00 -1.000000e+00
      0.000000e+00 0.000000e+00 1.000000e+00 -1.000000e+00
      0.000000e+00 0.500000e+00 1.000000e+00 -1.000000e+00
      0.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00
      1.000000e+00 -1.000000e+00 0.000000e+00 0.000000e+00
      1.000000e+00 -0.500000e+00 0.000000e+00 0.000000e+00
      1.000000e+00 0.000000e+00 0.000000e+00 0.000000e+00
      1.000000e+00 0.500000e+00 0.000000e+00 0.000000e+00
      1.000000e+00 1.000000e+00 0.000000e+00 0.000000e+00
    </Stream>
  </Array>
</XSIL>
```

# B

# The xsil2graphics utility program

```
~/Applications/xmds/xmds-devel/examples$ xsil2graphics --help

xsil2graphics utility supplied with xmds version 1.6.5

Usage: xsil2graphics [options] infile
Options:
  infile:           required, the input xsil file
  -h/--help:        optional, display this information
  -m/--matlab:      optional, produce matlab output (default)
  -s/--scilab:      optional, produce scilab output
  -a/--mathematica: optional, produce mathematica 5.x ouput
  -e/--mathematica: optional, produce mathematica ouput
  -g/--gnuplot:     optional, produce gnuplot ouput
  -r/--R:           optional, produce R ouput
  -o/--outfile:     optional, alternate output file name
  -v/--verbose:     optional, verbose output

For further help, please see http://www.xmds.org
```

Most users will find this utility essential: it writes the output as `.dat` ascii files and writes a separate file which is executed from within a common plotting program in order to load in the data. Currently supported packages include matlab (http://www.mathworks.com), scilab (http://scilabsoft.inria.fr/), Mathematica (http://www.wolfram.com), gnuplot and R. If no output file name is specified then the last extension is removed from the input file name and an appropriate ending (`.m`, `.sci`, or `.nb`) is appended. To avoid confusion between data from different XSIL data containers (as there may have been more than one container within the input file specified) `xsil2graphics` appends all variable names with an integer specifying

which XSIL data container (in sequence) they came from.

There is an example plotting program for Mathematica in the examples directory. To use the output of `xsil2graphics` in matlab, one merely needs to call the name of the .xsil file without the .xsil extension. For example, for the nlse simulation, the output file is nlse.xsil and the command one uses in matlab is:

```
>> nlse
```

In scilab, for the same simulation, the command to use is:

```
-->exec('nlse.sci')
```

# C loadxsil.m utility script

## C.1 Name

loadxsil—load simulation data into matlab

## C.2 Synopsis

`loadxsil('<xsil_file>')`

## C.3 Description

Utility script bundled with **xmds**, used to load simulation output data (from the xsil data file) into matlab, where the results can be presented graphically.

To load data from the xsil file `data_file.xsil`, enter at the matlab command prompt:

```
>> loadxsil('data_file.xsil')
```

## C.4 Examples

At the matlab command prompt:

```
>> loadxsil('nlse.xsil')
```

loads the data contained in `nlse.xsil` into matlab

## C.5 Authors

Written by Paul Cochrane

## C.6 Bugs

No known bugs. However, the loadxsil script does not work in Matlab version 4.0 or below; it can only be used with Matlab version 5.0 and above. Users with Matlab 4.0 can use the **xsil2graphics** utility as a means to import data into Matlab.

## C.7 See also

xmds(1), xsil2graphics(1), http://www.xmds.org

## C.8 Copyright

# D

# Gnu General Public License

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy

of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly

provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

# Part VI

# Bibliography and Index

# Bibliography


[1] A. L. Garcia, *Numerical methods for physics* (Prentice-Hall, 1994).

[2] *XSIL documentation*, URL `http://www.cacr.caltech.edu/SDA/xsil/`.

[3] C. W. Gardiner, *Handbook of Stochastic Methods* (Springer Verlag, Berlin, 1997), 2nd ed.

[4] D. M. Ceperley, Rev. Mod. Phys. **71**, 438 (1999).

[5] E. Kreyszig, *Advanced Engineering Mathematics* (John Wiley & Sons, 1999).

[6] P. D. Drummond, Comp. Phys. Comm. **29**, 211 (1983).

[7] B. M. Caradoc-Davies, URL `http://www.physics.otago.ac.nz/research/uca/resources/bmcdthesis.html`.

[8] M. J. Davis, URL `http://www.`

[9] *XML syntax specification*, URL `http://www.w3.org/TR/2000/REC-xml-20001006`.

[10] E. R. Harold and W. S. Means, *XML in a Nutshell* (O'Reilly & Associates, Inc., Sebastopol, USA, 2002), 2nd ed.

[11] *Document Object Model specification*, URL `http://www.w3.org/DOM/`.

# Index