xSPDE: *extensible* Stochastic Partial Differential Equations

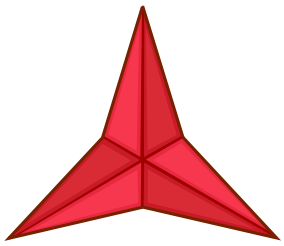
P. D. Drummond

September 14, 2015

xSPDE was developed at the
Centre for Quantum and Optical Science,
Swinburne University of Technology,
Melbourne, Victoria, AUSTRALIA.
Documented version: xSPDEv1.02

Thanks for much valuable feedback and many useful suggestions from:

Bogdan Opanchuk, Rodney Polkinghorne, Simon Kiesewetter, Laura Rosales-Zarate, King Ng and Run Yan Teh.



Contents

1 Introduction

1.1 xSPDE: a stochastic toolbox

The xSPDE code is an extensible Stochastic Partial Differential Equation solver. It is a **stochastic toolbox** for constructing simulations, which is applicable to many stochastic problems[1]. It has a modular design which can be changed to suit different applications, and includes strategies for calculating errors. At a basic level just one or two lines of input are enough to specify the equation. For advanced users, the entire architecture is open and extensible in numerous ways.

Versions with an .m ending are written in Matlab, an interpreted scientific language of The Mathworks Inc. This version is best regarded as a prototyping platform. A code for new applications can be quickly developed and tested. This will not be quite as fast as a dedicated code but can be written easily and *understandably*.

The xSPDE logo is a three-pointed star that symbolizes that the code is suitable for all three domains: ordinary, partial or stochastic equations. Coincidentally, the logo is also similar to the three-pointed star from a Mercedes 280SE, renowned for its advanced automotive engineering.

Readers of this document may also wish to try XMDS [2], and its successor, XMDS2 [3, 4], which are similar programs using XML input files.

1.2 Applications

Stochastic equations are equations with random noise terms [1]. They occur in in many fields of science, engineering, economics and other disciplines. xSPDE can solve both ordinary and partial differential stochastic equations. These include partial spatial derivatives, like the Maxwell or Schrödinger equations.

There are many equations of this type, and xSPDE can treat a wide range. It has a configurable functional design. The general structure permits drop-in replacements of the functions provided. Different simulations can be carried out sequentially, to simulate the various stages in an experiment or other process.

The code supports parallelism at both the vector instruction level and at the thread level, using Matlab matrix instructions and the parallel toolbox. It calculates averages of arbitrary functions of any number of complex or real fields. It uses sub-ensemble averaging and extrapolation to obtain accurate error estimates.

Note: xSPDE is distributed without any guarantee, under the MIT open-source license. Please test it yourself before use.

2 Interactive xSPDE

All xSPDE simulations require parameters stored in an input structure used by the xSPDE toolbox. Inputs have default values, which can be changed by typing parameters into the *in* structure. A complete list of parameters and their uses is given in Chapter 4, with a summary table in the Appendix.

The xSPDE toolbox function introduced in this chapter is:

- **xspde(in)**, the combined simulation with graphics function.

Parameters used in this chapter are:

Label	Type	Default value	Description
fields	integer	1	Number of stochastic variables
noises	integer	*1*	Number of noises
olabels	string	*'a_ 1'*	Observable labels
da	function	*0*	The stochastic derivative
initial	function	0	Function to initialize variables
observe	function	*a(1,:)*	Observable function

2.1 Stochastic equations

An ordinary stochastic equations [5, 6] for a real or complex vector $\mathbf{a}$ is:

$$\frac{\partial \boldsymbol{a}}{\partial t} = \dot{\boldsymbol{a}} = \mathbf{A}\left(\boldsymbol{a}\right) + \underline{\mathbf{B}}\left(\boldsymbol{a}\right)\boldsymbol{\zeta}(t). \tag{2.1}$$

Here$\mathbf{A}$ is a vector, $\underline{\mathbf{B}}$ a matrix, and $\boldsymbol{\zeta}$ is a real noise vector such that:

$$\left\langle \zeta_i\left(t\right)\zeta_j\left(t\right)\right\rangle \quad = \quad \delta\left(t-t'\right)\delta_{ij}$$

2.1.1 Running xSPDE interactively

To simulate a stochastic equation interactively, make sure Matlab path is pointing to the xSPDE folder and type *clear* to clear old data.

Next, enter the xSPDE inputs and functions into the command window, as follows:

$$\begin{aligned} in.label1 &= parameter1 \\ in.label2 &= \ldots \\ in.da &= @(a,z,r)\ (\dot{a}) \\ data &= xspde(in) \end{aligned}$$

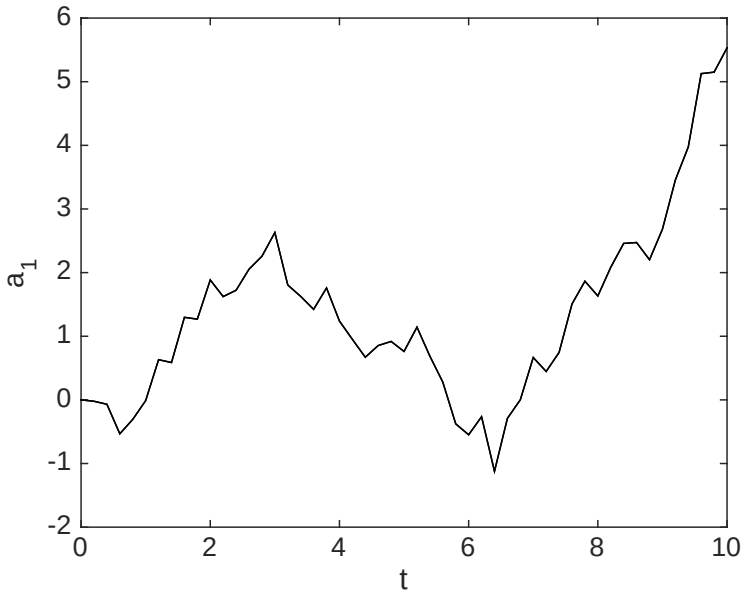


Figure 2.1: The simplest case: a random walk.

- The notation $in.label = parameter$ defines metadata in the structure in.
- The notation @(..) is the Matlab shorthand for a function.
- In xSPDE, a is the stochastic variable, z the random noise, and r the parameters.

2.1.2 The random walk

The first example is the simplest possible stochastic equation:

$$\dot{a} = \zeta(t)\,, \tag{2.2}$$

with a complete xSPDE script in Matlab below, and output in Fig (2.1).

```
in.da=@(a,z,r) z
data = xspde(in);
```

- **Here $in.da$ defines the derivative function, with z the noise.**

2.1.3 Laser quantum noise

Next we treat a model for the quantum noise of a single mode laser:

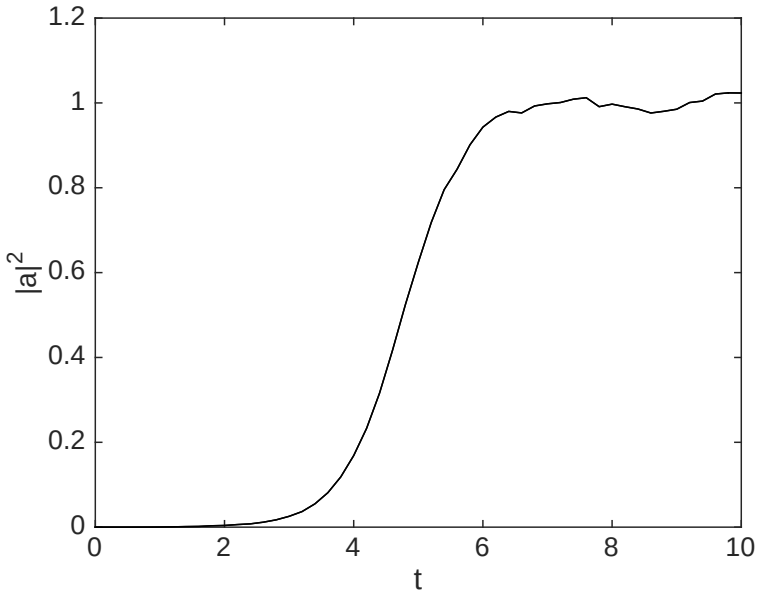


Figure 2.2: Simulation of the stochastic equation describing a laser turning on.

$$\dot{a} = \left(1 - |a|^2\right) a + b\zeta(t) \tag{2.3}$$

where $\zeta = (\zeta_1 + i\zeta_2)$, so that:

$$\langle \zeta(t)\zeta^*(t')\rangle = 2\delta\left(t - t'\right) . \tag{2.4}$$

Here the coefficient b describes the quantum noise of the laser, and is inversely proportional to the equilibrium photon number. An interactive xSPDE script in Matlab is given below, for the case of $b = 0.01$, with an output graph in Fig (2.2).

```
in.noises=2;
in.observe = @(a,r) abs(a)^2;
in.olabels = '|a|^2';
in.da=@(a,z,r) (1-abs(a)^2)*a+0.01*(z(1)+i*z(2));
xspde(in)
```

Note that:

- *in.noises* **is the number of noises,**
- *in.observe* **is the graphed function,**
- *in.olabels* **gives the axis label.**

2.2 Ito and Stratonovich equations

The xSPDE toolbox is primarily designed to treat Stratonovich equations [1], which are the broad-band limit of a finite band-width random noise equation, whose derivatives are evaluated at the midpoint in time of a time-step.

An equivalent type of of stochastic equation is the Ito form. This is written in a similar way to a Stratonovich equation, except that this corresponds to a limit where derivatives are evaluated at the start of each step. To avoid confusion, we can write an Ito equation as a difference equation:

$$d\mathbf{a} = \mathbf{A}^I[\mathbf{a}] + \underline{\mathbf{B}}[\mathbf{a}] \cdot d\boldsymbol{w}(t). \tag{2.5}$$

Here $\langle dw_i(\boldsymbol{x})\, dw_j(\boldsymbol{x}')\rangle = \delta_{ij} dt$. When $\mathbf{B}$ is not a constant, the Ito drift term is different to the Stratonovich one. This difference occurs because the noise term is non-differentiable. The relationship is that

$$A_i \;=\; A_i^I - \frac{1}{2}\sum_{j,m} \frac{\partial B_{ij}}{\partial a_m} B_{mj}\,. \tag{2.6}$$

2.2.1 Financial calculus

A well-known Ito stochastic equation is the Black-Scholes equation, used to price financial options. It describes the fluctuations in a stock value:

$$da = \mu a\, dt + a\sigma\, dw, \tag{2.7}$$

where $\langle dw^2\rangle = dt$. Since the noise is multiplicative, the equation is different in Ito and Stratonovich forms of stochastic calculus. The corresponding Stratonovich equation, as used in xSPDE is:

$$\dot{a} = \left(\mu - \sigma^2/2\right) a + a\sigma\zeta(t). \tag{2.8}$$

An interactive xSPDE script in Matlab is given below with an output graph in Fig (2.3), for the case of a volatile stock with $\mu = 0.1$, $\sigma = 1$. Note the spiky behaviour, typical of multiplicative noise, and also of the risky stocks in the small capitalization portions of the stock market.

```
in.initial=@(v,r) 1
in.da=@(a,z,r) -0.4*a+a*z
xspde(in)
```

- **Note that** *in.initial* **describes the initialization function.**
- The first argument of $@(v,r)$ is v, an initial random variable.
- **The error-bars are estimates of step-size error.**
- Errors can be reduced by using more time-steps: see Chapter (3).

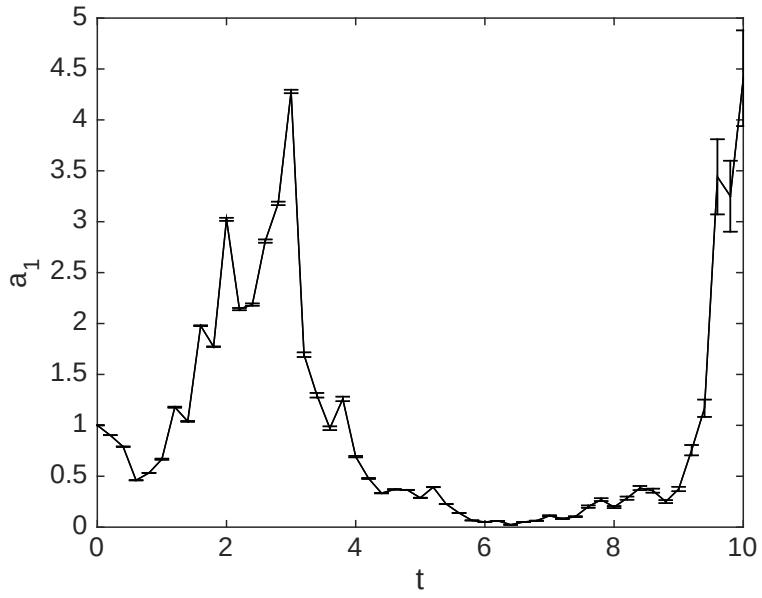


Figure 2.3: Simulation of the Black-Scholes equation describing stock prices.

2.3 Stochastic partial differential equations

More generally, xSPDE solves [7] a stochastic partial differential equation for a complex vector field defined in space-time dimension $d = 1 - 4$, written in differential form as

$$\frac{\partial \mathbf{a}}{\partial t} = \mathbf{A}\left[\mathbf{a}\right] + \underline{\mathbf{B}}\left[\mathbf{a}\right] \cdot \boldsymbol{\zeta}(t) + \underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right] \cdot \mathbf{a}\,. \tag{2.9}$$

Here $\mathbf{a}$ is a real or complex vector or vector field. The initial conditions are arbitrary functions. $\mathbf{A}\left[\mathbf{a}\right]$ and $\underline{\mathbf{B}}\left[\mathbf{a}\right]$ are vector and matrix functions of $\mathbf{a}$, $\underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right]$ is a matrix of linear terms and derivatives, diagonal in the vector indices, and $\boldsymbol{\zeta} = \left[\boldsymbol{\zeta}^x, \boldsymbol{\zeta}^k\right]$ are real delta-correlated noise fields such that:

$$\begin{aligned} \left\langle \zeta_i^x\left(t,\boldsymbol{x}\right)\zeta_j^x\left(t,\boldsymbol{x}'\right)\right\rangle &= \delta\left(\boldsymbol{x}-\boldsymbol{x}'\right)\delta\left(t-t'\right)\delta_{ij} \\ \left\langle \zeta_i^k\left(t,\boldsymbol{k}\right)\zeta_j^k\left(t,\boldsymbol{k}'\right)\right\rangle &= f(\mathbf{k})\delta\left(\boldsymbol{k}-\boldsymbol{k}'\right)\delta\left(t-t'\right)\delta_{ij}. \end{aligned} \tag{2.10}$$

Transverse boundary conditions are assumed periodic. The term $\underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right]$ may be omitted if $d = 1$, as there are no space dimensions. Here $f(\mathbf{k})$ is an arbitrary momentum filter, for correlated noise.

To treat stochastic partial differential equations or SPDEs, the equations are divided into two separate parts. The first two terms are essentially an ordinary stochastic equation, while the last term gives an exactly soluble linear partial differential equation, so that:

$$\frac{\partial \mathbf{a}}{\partial t} = \underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right] \cdot \mathbf{a} \tag{2.11}$$

The *interaction picture* is a moving reference frame used to solve the linear part of the equation exactly, defined by an exponential transformation. This is carried out internally by matrix multiplications and Fourier transforms.

In more detail, in Fourier space, if $\tilde{\mathbf{a}}(\mathbf{k}) = \mathcal{F}[\mathbf{a}(\mathbf{x})]$ is the Fourier transform of $\mathbf{a}$, we simply define:

$$\tilde{\mathbf{a}}(\mathbf{k}, dt) = \mathcal{P}(\mathbf{k}, dt)\, \tilde{\mathbf{a}}_I(\mathbf{k}, dt) \tag{2.12}$$

where the propagation function can be written intuitively as $\mathcal{P} = \exp[\underline{\mathbf{L}}(\mathbf{D})dt]$, where $\mathbf{D} = i\mathbf{k} \sim \nabla$. The function $\underline{\mathbf{L}}(\mathbf{D})$ is input using the xSPDE function **xlinear**. With this definition, at each step the equation that is solved can be re-written in a more complicated looking, but actually more readily soluble form as:

$$\frac{\partial \mathbf{a}_I}{\partial t} = \mathcal{D}\left[\mathcal{F}^{-1}\mathcal{P}(\mathcal{F}\mathbf{a}_I)\right] \tag{2.13}$$

The total derivative in the interaction picture is the xSPDE function **xda:**

$$\dot{\mathbf{a}}_I = \mathbf{A} + \underline{\mathbf{B}}\boldsymbol{\zeta} \tag{2.14}$$

where usually $\mathbf{A}, \underline{\mathbf{B}}$ are evaluated at the midpoint which is the origin in the interaction picture. For convenience, the final output is calculated in the original picture, so there are interaction picture (IP) transformations at each time-step.

2.3.1 Symmetry breaking

Including space-time dimensions with $d = 3$, an example of a SPDE is the stochastic Ginzburg-Landau equation. This describes symmetry breaking, as the system develops a spontaneous phase which can vary spatially as well. The model is widely used in fields ranging from lasers to magnetism, superconductivity, superfluidity and even particle physics:

$$\dot{a} = \left(1 - |a|^2\right) a + b\zeta(t) + ic\nabla^2 a \tag{2.15}$$

where

$$\langle \zeta(x)\zeta^*(x')\rangle = 2\delta(t - t')\,\delta(x - x'). \tag{2.16}$$

The full xSPDE script is given below, for parameter values of $b = 0.001$ and $c = 0.01$, with the output graphed in Fig (2.4) .

```
in.noises=2;
in.dimension=3;
in.steps=10;
in.linear = @(D,r) i*0.01*(D.x.^2+D.y.^2);
in.observe = @(a,~) abs(a).^2;
in.olabels = '|a|^2';
in.da=@(a,z,~) (1-abs(a(1,:)).^2).*a+0.001*(z(1,:)+i*z(2,:));
xspde(in)
```

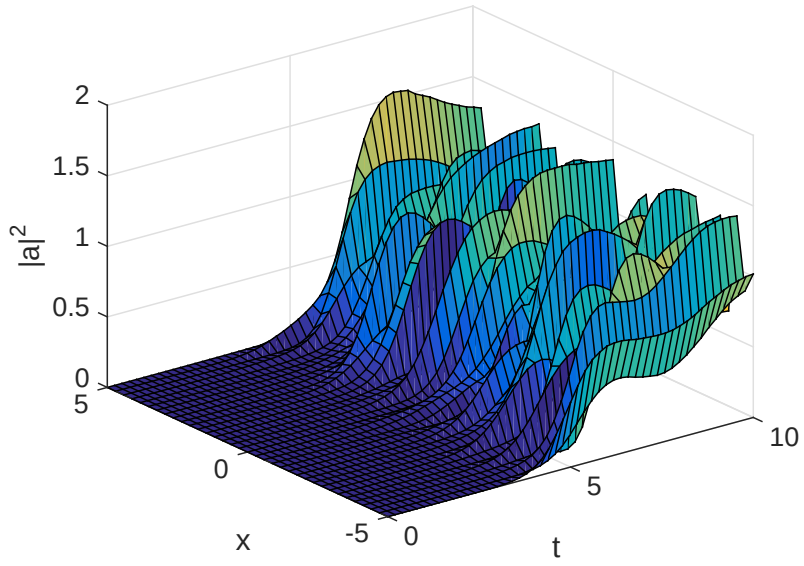


Figure 2.4: Simulation of the stochastic equation describing symmetry breaking in two dimensions. Spatial fluctuations are caused by the different phase-domains that interfere. The graph obtained here is projected onto the $y = 0$ plane.

Here:

- *in.dimension* **is the space-time dimension, with an** $x - t$ **plot given here.**
- *in.steps* **gives the integration steps per plot-point, for improved accuracy.**
- *in.linear* **is the linear operator - an imaginary laplacian -**
- $D.x$ **indicates a derivative operation,** $\partial/\partial x$**.**
- $-5 < x < 5$ **is the default xSPDE coordinate range.**
- **The .* notation is used, as fields require element-wise multiplication.**

3 xSPDE averaging and projects

The xSPDE toolbox functions introduced in this chapter are:

- **xsim(in)**, the xSPDE simulation function.
- **xgraph(data, in)**, the SPDE graphics function.

New input parameters introduced in this chapter:

Label	Type	Default value	Description
ensembles	vector	*[1 1 1]*	Ensembles used for averaging
name	string	' '	Simulation name
file	string	"	File-name for HDF5 or Matlab data file
ranges	vector	10	Range of coordinates in [t,x,y,z]
points	vector	51	Output lattice points in [t,x,y,z]

3.1 Using xsim and xgraph

Suppose that you are not happy with the graphs in an interactive session, and want to alter them by adding a heading or some other change. For long simulations, it may be inconvenient to re-run everything. This is easy, provided the data is in the current workspace.

For example, suppose the Kubo simulation is run interactively, using an output specification so that the *data* file is stored locally in your Matlab workspace. After editing some inputs, the job can be repeated, with data saved to the local workspace, by running

$$[e, data] = xspde(in)\,.$$

Alternatively, if you just want the data, and no graphs, use:

$$[e, data] = xsim(in)\,.$$

Next, you should know the object label you wish to change. Note that the **xsim** and **xgraph** program inputs are either structure objects containing data for one simulation, or else cell arrays of structures with many simulations in sequence, which are treated later. You can use either in **xSPDE**.

To change the project name for the graph headings, input:

$$in.name =' My\,new\,project',$$

then replot the data using

$$xgraph(data, in)\,.$$

More graphics information can be added, for example using

$$in.olabels =' x'$$

which inputs this additional graphics data to the *in* structure to xSPDE. Just plot again, using the same instructions.

3.2 Ensembles in xSPDE

Averages over stochastic ensembles are the specialty of xSPDE, which requires specification of the ensemble size. To get an average, an ensemble size is needed, to obtain more trajectories in parallel.

Ensembles are specified in three levels to allow maximum resource utilization, so that:

$$in.ensembles = [ensembles(1), ensembles(2), ensembles(3)]\,.$$

The first, *ensembles* (1), gives within-thread parallelism, allowing vector instruction use for single-core efficiency. The second, *ensembles* (2), gives a number of independent trajectories calculated serially.

The third, *ensembles* (3), gives multi-core parallelism, and requires the Matlab parallel toolbox. This improves speed when there are multiple cores, and one should optimally put *ensembles* (3) equal to the available number of CPU cores.

The *total* number of stochastic trajectories or samples is

$$ensembles(1) \times ensembles(2) \times ensembles(3)\,.$$

However, either *ensembles*(2) or *ensembles*(3) are required if sampling error-bars are to be calculated, owing to the sub-ensemble averaging method used in xSPDE to calculate sampling errors accurately.

3.2.1 Random walk with averaging

To demonstrate this, try adding some more trajectories, points and an output label to the *in* parameters of the random walk example:

```
in.da=@(a,z,r) z;
in.ensembles = [500,20];
in.points =101;
in.olabels = '<a_1>';
xspde(in)
```

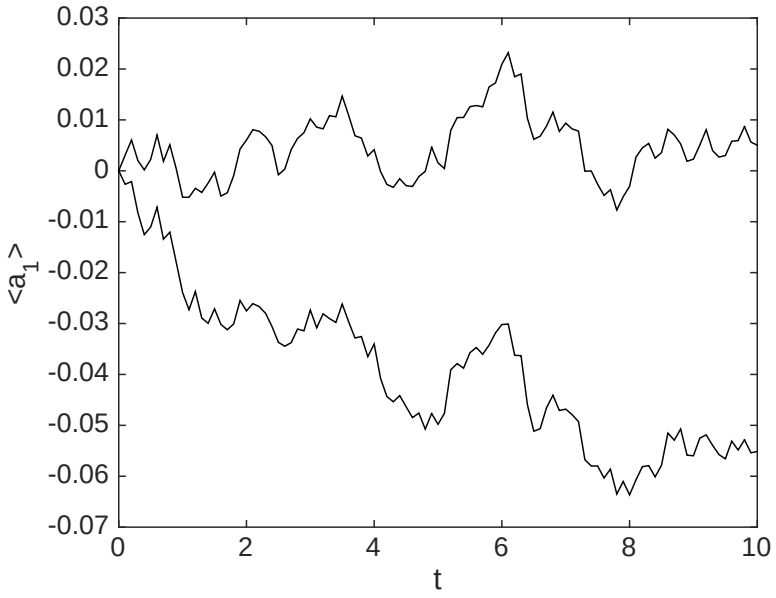


Figure 3.1: The average random walk with 10^4 trajectories.

You will see Fig (3.1) appearing.

This looks like the earlier graph, but check the vertical scale. The *in.ensembles* = $[100, 100]$ input gives an average over 10^4 random trajectories. Therefore the sampling error is accordingly reduced by a factor of 100. The two lines plotted are the upper and lower one standard deviation limits.

The more detailed structure of the random walk is due to having more time-points. Of course, $\langle a \rangle = 0$ in the ideal limit.

Note that:

- *in.ensembles* **is the number of trajectories averaged over**
- *in.points* **is the number of time-points integrated and graphed**

3.3 xSPDE projects

An XPDE session can either run simulations interactively, described in Chapter 2, or else using a function file called a project file. This allows xSPDE to run in a batch mode, as needed for longer projects which involve large ensembles. In either case, the Matlab path must include the xSPDE folder.

A minimal xSPDE project function is as follows:

$$function \quad = \quad project.m$$

$$\begin{aligned} in.label1 &= parameter1 \\ in.label2 &= parameter2 \\ &\dots \\ xspde(in) & \\ end & \end{aligned}$$

For standard graph generation, the script input or project function should end with the combined function **xspde(in)**. Alternatively, to generate simulation data and graphs separately, the function *xsim* runs the simulation, and *xgraph* makes the graphs. The two-stage option is better for running batch jobs, which you can graph at a later time. See the next chapter for details.

After preparing a project, type the project name into the Matlab interface, or click on the Run arrow above the editor window.

In summary:

- For medium length simulations with more control, use a function file whose last executable statement is **xspde(in).**

3.3.1 Kubo project

To get started on more complex stochastic programs, we next simulate the Kubo oscillator, which is a stochastic equation with multiplicative noise. It uses the Stratonovich stochastic calculus. It corresponds to an oscillator with a random frequency, with difference equation:

$$\dot{a} = ia\zeta \tag{3.1}$$

To simulate this, one can use a file, *Kubo.m*, which also contains definitions of user functions.

```
function [e] = Kubo()
in.name = 'Kubo oscillator';
in.ensembles = [400,16];
in.initial = @(v,~) 1+0*v;
in.da = @(a,z,r) i*a.*xi;
in.olabels = {'<a_1>'};
e = xspde(in);
end
```

The resulting graph is given in Fig (3.2), including upper and lower one standard deviation sampling error limits to indicate the accuracy of the averages. This is obtained on inputting the second number in the ensembles vector, to allow sub-ensemble averaging and sampling error estimates. Note that .∗multiplication must be used because the first ensemble is stored as a matrix, to improve speed.

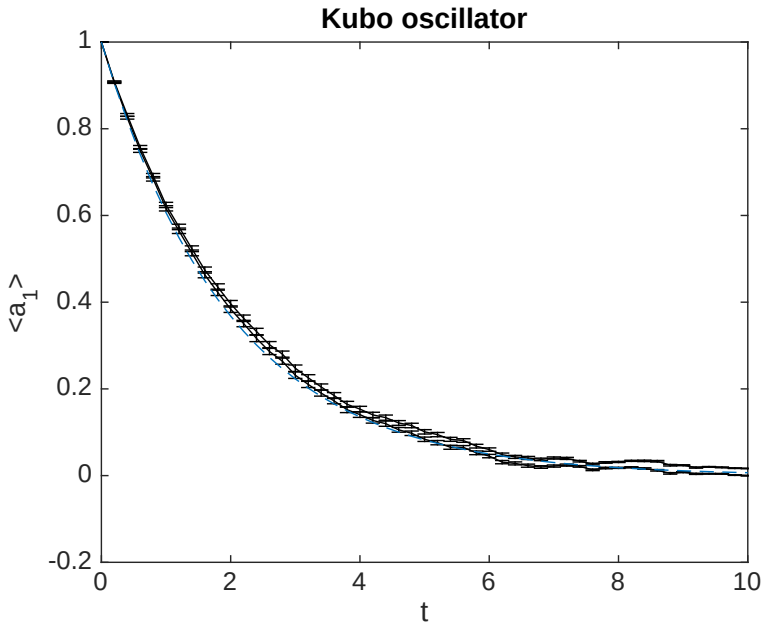


Figure 3.2: The amplitude decay of a Kubo oscillator.

The other input parameters are not specified explicitly. Default values are accessed from the ***inpreferences*** function.

Here we note that;

Kubo defines the parameters and function handles, then runs the simulation.

in.name gives a name to identify the project.

in.ensembles specifies 400 samples in a parallel vector, repeated 16 times in series.

in.initial initializes the input to ones; the noise v is used as it has the same lattice dimension as the a field.

in.da is the function, $da/dt = ia\zeta$, that specifies the equation being integrated.

in.olabels is a cell array with a label for the variable a that is averaged.

xspde(in) runs the simulation and graphics program using data from the **in** structure.

The function names can point to external files instead of those in the project file itself. This is useful when dealing with complex projects, or if you just want to change one function at a time. As no points or ranges were specified, here, default values of 51 points and a range of 10 are used.

3.4 Data files and batch jobs

It is often inconvenient to work interactively, especially for large simulations. To save data is very useful. This is not automatic: to create a data file, you must enter the filename - either interactively or a bath file - before running the simulation, using the $in.file = filename$ input.

The **xSPDE** program allows you to specify a file name that stores data in either standard HDF5 format, or in Matlab format. It also gives multiple ways to edit data for either simulations or graphs. A simple interactive workflow is as follows:

- Create the metadata *in*, and include a file name.
- Run the simulation with **xsim(in).**
- Run **xgraph(",in)**, and the data will be accessed and graphed.

3.4.1 Saving data files

In greater detail, first make sure you have a writable working directory with the command *cd* $\sim$. Next, specify the filename using the *in.file='name.ext'* inputs, and run the simulation. All calculated *data* as well as the input metadata from the *in* object is stored.

For example,

$$in.file = filename.mat$$

gives a Matlab data file - which is the simplest to edit.

Alternatively,

$$in.file = filename.h5$$

gives an international standard HDF5 data file, useful for exchanging data with other programs.

To reload and reanalyze any previously saved Matlab simulation data, say $Kubo.mat$, at a later time, there are two possible approaches, described below.

3.4.2 Graphing saved data

If the *in* metadata is still available in an interactive session, just type

$$xgraph('', in)\,,$$

which tells *xgraph* to use the file-name already present in *in*.

More generally, one can use a file name directly in *xgraph*, which works with any file type. Once the data is saved in a file by running *xsim* or *xspde* with an input $filename$, just type:

$$xgraph\,(filename.mat)$$

or for HDF5 files,

$$xgraph\,(filename.h5)$$

to replot the resulting data.

Note that you can use *xgraph* with either Matlab or HDF5 file data inputs, and without having to specify the *in* structure. This metadata is automatically saved with the data in the output file. This approach has the advantage that many simulations can be saved and then graphed later.

3.4.3 Editing saved data

If the saved data was a Matlab file, one can load the simulated data and metadata by typing, for example,

$$load\,Kubo\,.$$

Results can easily be replotted interactively, with changed input and new graphics details, using this method. This approach loads all the relevant saved data into your work-space.

Hence one can easily edit and re-edit the graphics inputs in the *in* structure, by using:

$$xgraph(data, in).$$

To change cell contents for a sequence, be aware that sequence inputs are stored in cell arrays with curly bracket indices.

3.4.4 Combining saved data with new metadata

If the graphs generated from saved data files need changing, some new input specifications may be needed.

To combine an old, saved data file, say $'Kubo.mat'$ with a new input specification *in* you have just created, type

$$xgraph\,('Kubo.mat', in)\,,$$

or if the data was saved in an $HDF5$ file, it is:

$$xgraph\,('Kubo.h5', in)\,.$$

Provided the *in* structure is present, and includes a valid file-name, you can also use:

$$xgraph\,('', in)\,.$$

With this input the *xgraph* program automatically knows that it should look for the file-name in the *in* metadata.

In all cases the new *in* metadata is combined with the old metadata. Any new input metadata takes precedence over old, filed metadata.

This allows fonts and labels, for example, to be easily changed - without having to re-enter all the simulation input details.

Note that $xgraph\,('')$ is obviously invalid.

3.5 Sequential integration

Sequences of calculations are available simply by adding a sequence of inputs to xSPDE, representing changed conditions or input/output processing. These are combined in a single file for data storage, then graphed separately. The results are calculated over specified ranges, with it's own parameters and function handles. In the current version of xSPDE, the numbers of ensembles must be the same throughout.

The initialization routine for the first fields in the sequence is called **initial**, while for subsequent initialization it is called **transfer.** The sequential initialization function has four input arguments, to allow noise to be combined with previous field values and input arguments, as may be required in some types of simulation. This is described in the next chapter.

In many cases, the default transfer value - which is to simply reuse the final output of the previous set of fields - is suitable. To help indicate the order of a sequence, a time origin can be included optionally with sequential plots, so that the new time is the end of the previous time, if this is required.

Suppose the project has a sequence of two simulations, with input structures of $in1$, $in2$. To run this and store the data locally, just type:

$$[e, data] = xspde(\{in1, in2\})\,.$$

To change the file headers, at a later stage, type:

$$\begin{aligned} in1.name &= 'My\,first\,simulation' \\ in2.name &= 'My\,next\,simulation'. \end{aligned}$$

This method requires that the $data$ and $in1, in2$ are already loaded into your Matlab work space so they can be edited.

Next, simply replot the data using

$$xgraph(data, \{in1, in2\})\,.$$

3.6 xSPDE hints

- When using xSPDE, it is a good idea to first run the batch test script, Batchtest.m. This will perform simulations of different types, and report an error score. The final error score ought to match the number in the script comments, to show your installation is working correctly.
- xSPDE also tests your parallel toolbox installation. If you have no license for this, just omit the third ensemble setting, so that the parallel option is not used.
- To create a project file, it is often easiest to start with an existing project function with a similar equation. There are a number of these distributed with xSPDE, and these are included in the Batchtest examples.

- Just as in interactive operation, the simulation parameters and functions for a batch job are defined in the structure **in**. The parameters include *function handles* that point to user specified functions, which give the initial values, derivative terms and quantities measured. The function handles can point to any function declarations in the same file or Matlab path.
- Graphics parameters and a comparison function are also defined in the structure **in**. In each case there are default parameters in a preference file, but the user inputs will be used first if included.

The general workflow is as follows:

Create a project file, using an existing example as a template

Decide whether you want to generate graphs now (*xSPDE*) or later (*xsim* and *xgraph*).

Edit the project file parameters and functions

Check that the Matlab path includes the xSPDE folder

Click *Run*

Save the output graphs that you want to keep

More details and examples will be given in later sections!

4 Input parameters

This chapter summarizes the input parameters and functions.

4.1 xSPDE functions

[e,data] = xspde(input)

This is the combined xSPDE function. It accepts a simulation sequence, *input*. As well as generating graphs, it returns a combined error *e,* and the *data* array. It calls the functions **xsim** and **xgraph**, explained below.

[e,data] = xsim(input)

This is the xSPDE simulation function. It accepts a simulation sequence in the *input* cell array and returns error estimates, together with a cell array of simulated *data*. The data are: mean values of functions, error bars and sampling errors. This can be run as a stand-alone function.

ec = xgraph(data,input)

This is the xSPDE graphics function. It takes computed simulation *data* and *input* parameter specifications. It plots graphs, and returns the maximum difference *ec* from comparisons. The *data* should have as many cells as *input* cells, for sequences. If $data = \{''\}$, then an HDF5 data file will be read using the file-name specified in *input*. If $data = \{'filename.h5'\}$ or $data = \{'filename.mat'\}$ then the specified file is read as data. Note that $.h5$ indicates an $HDF5$ file format, and $.mat$ indicates a Matlab internal file format.

Simulation parameters

For each simulation in the *input* sequence, the input and functions are specified as a data structure, *in*. These can be entered either interactively or as part of a simulation function file. The function file approach allows recycling and editing, so it is better for a large project.

There are extensive default preferences to simplify the inputs. If any inputs are omitted, there are default values which are set by inpreferences in all cases. These defaults are changed by editing the inpreferences function. The *xgrpreferences* function is used to supply graphics default values.

For vector or cell inputs, an input shorter than required is padded to the right using default values.

4.2 Input parameters and user functions

A sequence of simulations is obtained from inputs in a cell array, as $input = \{in1, in2, ..\}$. The input parameters of each simulation in the sequence are specified in a Matlab structure. If there is one simulation, just one structure can be input, without the braces.

The standard form of each parameter value is:

$$in.label = parameter$$

The inputs are scalar or vector parameters or function handles. Quantities relating to graphed averages are cell arrays, indexed by the graph number. The available inputs, with their default values in brackets, are as follows.

4.2.1 Parameters

in.name (=' '): Name used to label simulation, usually corresponding to the equation or problem solved. This can be added or removed from graphs using the ***in.headers*** Boolean variable, as explained in the section on graphics parameters.

$$in.name =' your\, project\, name'$$

in.dimension *(=1):* The total space-time dimension is labelled, unsurprisingly,

$$in.dimension = 1 \ldots 4.$$

in.fields *(=1):* These are real or complex variables stored at each lattice point, and are the independent variables for integration. The fields are vectors that can have any dimension.

$$in.fields = 1, 2, \ldots$$

in.randoms *(=fields)*: This gives the number of random fields generated per lattice point for the initial noise, in coordinate and momentum space. Set to zero ($in.randoms = 0$) for no random fields. Random fields can be correlated either in ordinary or momentum spaces. The second input is the dimension of random fields in momentum space. It can be left out if zero. Note that $n.random = randoms(1) + randoms(2)$:

$$in.randoms = [randoms\,(1)\,, randoms\,(2)] \geq 0\,.$$

in.noises *(=fields)*: This gives the number of stochastic noises generated per lattice point for both the initial noise and the integration noise, in coordinate and momentum space. Set to zero ($in.noises = 0$) for no noises. This is the number of *rows* in the noise-vector. Noises can be correlated either in ordinary or momentum spaces.

The second input is the dimension of noises in k-space. It can be left out if zero. Note that $n.noise = noises(1) + noises(2)$:

$$in.noises = [noises\,(1)\,, noises\,(2)] \geq 0\,.$$

in.ranges *(=[10,10,..]):* Each lattice dimension has a coordinate range, given by:

$$in.ranges = [ranges\,(1)\,, \ldots\,, ranges\,(dimension)]\,.$$

In the temporal graphs, the first coordinate is plotted over $0 : ranges(1)$. All other coordinates are plotted over $-ranges(n)/2 : ranges(n)/2$. The default value is 10 in each dimension.

in.points *(=[49,35,...35]):* The rectangular lattice of points plotted for each dimension are defined by a vector giving the number of points in each dimension:

$$in.points = [points\,(1)\,, \ldots\,, points\,(dimension)]\,.$$

The default values are simply given as a rough guide for initial calculations. Large, high dimensional lattices take more time to integrate. Increasing $in.points$ improves graphics resolution, and gives better accuracy in each relevant dimension as well, but requires more memory. Speed is improved when the lattice points are a product of small prime factors.

in.steps *(=1)*: Number of time-steps per plotted point. The total number of integration steps in a simulation is therefore $steps \times (points(1) - 1)$. Thus, *steps* can be increased to improve the accuracy, but gives no change in graphics resolution. **Increase** steps to give a **lower** time-discretization error:

$$in.steps = 1, 2, \ldots$$

in.ensembles *(=[1,1,1])*: Number of independent stochastic trajectories simulated. This is specified in three levels to allow maximum parallelism. The first gives within-thread parallelism, allowing vector instructions. The second gives a number of independent trajectories calculated serially. The third gives multi-core parallelism, and requires the Matlab parallel toolbox. Either $ensembles(2)$ or $ensembles(3)$ are required if sampling error-bars are to be calculated.

$$in.ensembles = [ensembles\,(1)\,, ensembles\,(2)\,, ensembles\,(3)] \geq 1$$

The *total* number of stochastic trajectories or samples is $ensembles(1) \times ensembles(2) \times ensembles(3)$.

in.transforms *(=\{0\})*: **Cell array** that defines the different transform spaces used to calculate field observables. This has the structure

$$in.transforms\{n\} = [t\,(1)\,, \ldots t(4)] \geq 0$$

There is one transform vector per observable. The $j-th$ index, $t(j)$, indicates a Fourier transform on the j-th axis. The normalization of the Fourier transform is such that the $k = 0$ value in momentum space corresponds to the integral over space, with an additional factor of $1/\sqrt{2\pi}$. This gives a Fourier integral which is symmetrically normalized in ordinary and momentum space. The Fourier transform is such that $k = 0$ is the *central* value.

in.olabels (={'a_1 ',..}): **Cell array** of labels for the graph axis observable functions. These are text labels that are used on the graph axes. The default value is a_1 if the default observable is used, otherwise it is blank. This is overwritten by any subsequent label input when the graphics program is run:

$$in.olabels\{n\} =' string'$$

in.c: This starting letter is always reserved to store user-specified constants and parameters. All inputs - including 'c' data - are copied into the data files and also the lattice structure r. It is passed to user functions, and can be any data.

$$in.c = anything$$

4.2.2 Invariant inputs

The following can't be changed during a sequence in the current xSPDE version - the specified values for the first simulation will be used:

1. The extrapolation order
2. The number of ensembles (2)
3. The number of ensembles (3)

4.2.3 Input functions

A stochastic equation solver requires the definition of an initial distribution and a time derivative. In xSPDE, the time derivatives is divided up into a linear term including space derivatives, used to define an interaction picture, and the remaining derivatives. In addition, one must define quantities to be averaged over during the simulation, called graphs in xSPDE. These are all defined as functions, specified below.

in.initial (= @xinitial(v,r)) initializes the fields a for the first simulation in a sequence. The initial Gaussian random field variable, v, has unit variance if $dimension = 1$ or else is delta-correlated in space, with variance $1/(dx_2..dx_d) = 1/r.dV$ for d space-time dimensions. If specified in the input, ra has a first dimension of $n.random$, otherwise the default is $fields$. The default set by $x.initial$ is $a = 0$.

in.transfer (= @xtransfer(v,a0,r,r0)) initializes the fields a for subsequent calculations in a sequence. Otherwise, this function behaves in a similar way to $in.initial$. The function includes the previous field $a0$ and lattice $r0$. The default set by $xtransfer$ is $a = a0$.

in.da (= @xda(a,z,r)) calculates derivatives *da* of the equation. The noise vector, ζ, has variance $1/(dx_1..dx_d)$, for dimension $d \leq 4$, and a first dimension of $n.noise$ whose default value is $fields$. If specified by the two elements of the $noises$ vector, ζ can have a different first dimension from $fields$. This can also include noise correlated in momentum space.

in.linear (= @xlinear(D,r)) is a user-definable function which returns the linear coefficients L in Fourier space. This is a function of the differential operator D. The default is zero. Here D is a structure with components $D.x$, $D.y$, $D.z$. Each component has an array dimension the same as the coordinate lattice.

in.observe (= {@xobserve(a,r)}. **Cell array** of function handles that take the current field and returns a real observable o with dimension of $(1 \times n.lattice)$. The default observable is the first real field amplitude. Note the use of braces for cell arrays! One can also input these individually as $in.observe\{1\} = @(a,r)\, f(a,r)$, using an inline anonymous function. The total number of observe functions is stored internally as $in.graphs$. The fields a passed in the input are transformed according to the $in.transforms$ metadata.

in.rfilter(r) (=@xrfilter) returns the momentum-space filters for the input random terms. Each component has an array dimension the same as the coordinate lattice, that is, the return dimension is [r.randoms(2),r.n.lattice].

in.nfilter(r) (=@xnfilter) returns the momentum-space filters for the propagation noise terms. Each component has an array dimension the same as the coordinate lattice, that is, the return dimension is[r.noises(2),r.n.lattice].

4.3 Lattice, coordinates and time

Time and space

The default lattice for plotted output data is rectangular, with periodic boundary conditions in space, and:

$$dx(i) = \frac{in.ranges(i)}{in.points(i) - 1}$$

The time index is 1, and the space index i ranges from 2 to $in.dimension$. The maximum space-time dimension is $in.dimension = 4$, while $in.ranges(i)$ is the time and space duration of the simulation, and $in.points(i)$ is the total number of plotted points in the i-th direction.

Time is advanced in basic integration steps that are equal to or smaller than $dx(1)$, for purposes of controlling and reducing errors:

$$dt = \frac{dx(1)}{in.steps \times nc}$$

Here, *in.steps* is the minimum number of steps used per plotted point, and $nc = 1, 2$ is the check number. If $nc = 1$, the run uses coarse time-divisions. If $nc = 2$ the steps are halved in size for error-checking. Error-checking can be turned off if not required.

Functions

The xSPDE program is function oriented: user specified functions define initial conditions, equations and observables, amongst other things.

Function arguments are always in the following order:

- the field a or initial random variable v.
- the stochastic noise ζ or other fields
- non-field arguments.
- the grid structure r and any previous grid structure needed.

The first argument, a, is a real or complex vector field. This is a matrix whose first dimension is the field index. The second dimension is the lattice index.

The second argument, ζ, if needed, is a real random noise, usually abbreviated as z in the examples. This is a matrix whose first dimension is the noise index. The second dimension is the lattice index.

The last function argument is the current lattice structure, r. This contains data about the current integration lattice. The most important constants are $r.t$, the current time, and the space coordinates, $r.x$, $r.y$, $r.z$. Other data stored in the lattice structure is explained in later chapters.

Arrays

In all function calls, the variables used are matrices. The most important first dimension used is the field length $fields$. The second dimension in all arrays is the lattice index, with a length $n.lattice = ensembles(1) \times points(2) \times \ldots points(dimension)$. Here $ensembles(1)$ is the number of stochastic samples integrated as an array.

For reference, the field dimensions are:

- $a, da, L = [fields, n.lattice]$
- $v = [n.random, n.lattice]$
- $z = [n.noise, n.lattice]$
- $D.x$, $r.x$, $r.kx = [1, n.lattice]$
- $o = [1, n.lattice]$

Each observable is defined by a function in a cell array with length $n.graphs$.

4.4 Advanced parameters and functions

4.4.1 Advanced input parameters

More advanced input parameters, which don't usually need to be changed from default values, are as follows:

in.iterations *(=4)*: For iterative algorithms like the implicit midpoint method, the iteration count is set here, typically around $3-4$. Will increase the integration accuracy if set higher, but it may be better to increase *steps* if this is needed. With non-iterated algorithms, this input is not used:

$$in.iterations = 1, 2, \ldots$$

***in.*errorchecks** *(=2)*: This defines how many times the integration is carried out for error-checking purposes. If $errorchecks = 1$ there is one integration, but no checking at smaller time-steps. For error checking, set $errorchecks = 2$, which repeats the calculation at a shorter time-step - but with identical noise - to obtain the error bars, taking three times longer overall:

$$in.errorchecks = 1, 2$$

in.order *(=1):* This is the extrapolation order, which is **only** used if $errorchecks = 2$. The program uses the estimated convergence order to extrapolate to zero step-size, with reduced estimated error-bars. If $order = 0$, no extrapolation is used, which is the most conservative input. The default order is usually acceptable, especially when combined with the default midpoint algorithm, see next section. While any non-negative order can be input, the theoretical orders of the four preset methods used *without* stochastic noise terms are: xEuler = 1; xRK2 = 2; xMP = 2; xRK4 = 4. Allowed values are:

$$in.order \geq 0$$

in.seed *(=0)*: Random noise generation seed, for obtaining reproducible noise sequences. Only needed if $noises > 0$:

$$in.seed \geq 0$$

in.graphs *(*=number of observables): This gives the number of observables or graphs computed. The default is the length of the cell array of observable functions. Normally, this is not initialized, as the default is typically used. Can be used to suppress data averaging.

$$in.graphs \geq 0$$

in.print (=1): Print flag for output information while running xSPDE. If $print = 0$, most output is suppressed, while $print = 1$ displays a progress report, and $print = 2$ also generates a readable summary of the r lattice and gr graphics structures as a record.

$$in.print \geq 0$$

in.raw (=0): Flag for storing raw trajectory data. If this flag is turned on, raw trajectories are stored in memory and written to a file on completion. To make use of these, a file-name should be included!

$$in.raw \geq 0$$

in.origin (= [0,-ranges/2]): This displaces the graph origin for each simulation to a user-defined value. If omitted, all initial times in a sequence are zero, and the space origin is set to $-ranges/2$ to give results that are symmetric about the origin:

$$in.origin = [origin(1), \ldots origin\,(4)]$$

in.ipsteps (=1 for Euler and RK2, =2 for MP and RK4): This specifies the number of interaction picture steps needed in a full propagation time-step. Preferred values are chosen according to the setting of *in.step*. Can be changed for custom integration methods.

$$in.ipsteps = 1, 2, 3..$$

in.file (=' '): Matlab or *HDF5* file name for output data. Includes all data and parameter values, including raw trajectories if $raw = 1$. If not needed just omit this. A Matlab filename should end in *.mat*, while an HDF5 file requires the filename to end in .h5.

$$in.file = [origin(1), \ldots origin\,(4)]$$

4.4.2 Advanced input functions

Advanced input functions are user-definable functions which don't usually need to be changed from default values. They allow customization and extension of xSPDE. These are as follows:

in.grid(r) (=@xgrid) initializes the grid of coordinates in space.

in.noisegen(r) (=@xnoisegen) generates arrays of noise terms xi for each point in time.

in.randomgen(r) (=@xrandomgen) generates a set of random fields rf to initialize the fields simulated.

in.step(a,z,dt,r) (= @xMP) specifies the stochastic integration routine for a step in time dt and noise xi. It returns the new field **a** at space-time location **r**, given the old field as input, and interaction-picture propagator *r.propagator* which is part of the lattice structure. This can be set to any of the predefined stochastic integration routines provided with xSPDE, described in the Algorithms chapter. User-written functions can also be used. The standard method, xMP, is a midpoint integrator.

in.prop(a,r) (=@xprop) returns the fields propagated in the interaction picture, depending on the propagator array *r.propagator*.

in.propfactor(nc,r) (=@xpropfactor) returns the transfer array *r.propagator*, used by the *in.prop* function. The time propagated is a fraction of the integration time-step, *r.dt*. It is equal to $1/in.ipsteps$ of the integration time-step.

4.5 Graphics inputs and functions

The graphics parameters are also stored in the cell array *input* as a sequence of structures *in*. This only need to be input when the graphs are generated, and can be changed at a later time to alter the graphics output. A sequence of simulations is graphed from *input* specifications.

If there is one simulation, just one structure can be input, without the sequence braces. The standard form of each parameter value, which should have the *in*. structure label added, is:

$$in.label = parameter$$

If any inputs are omitted, there are default values which are set by the *xgrpreferences* function, in all cases except for the comparison function ***compare***. The defaults can be changed by editing the ***xgrpreferences*** function.

In the following descriptions, $n.graph$ is the total number of graphed variables of all types. The space coordinate, image, image-type and transverse data can be omitted if there is no spatial lattice, i.e, if the dimension variable is set to one.

4.5.1 Graphics functions

in.compare(t,in) This is a cell array of functions. Each takes the time or frequency vector and returns comparison results for a graphed observable, as a function of real values versus time or frequency. Comparison results are graphed with a dashed line, for the two-dimensional graphs versus time. There is no default function handle.

4.5.2 Graphics parameters

For uniformity, the graphics parameters are cell arrays, indexed over the graph number using braces {}. If a different type of input is used, like a scalar or matrix, xSPDE will attempt to convert the type. The axis labels are cell arrays, indexed over dimension.

Together with default values, they are:

in.font (= {18,..}): This sets the default font size for the graph labels.

$$in.font\{n\} > 0$$

in.minbar (= {0.01,..}): This is the minimum relative error-bar that is plotted.

$$in.minbar\{n\} \geq 0$$

in.images (={0,0,0,..}): This is the number of 3D, transverse o-x-y images plotted as discrete time slices. Only valid if $dimension > 2$. Note that, if present, the z-coordinate is set to its central value of $z = 0$, when plotting the transverse images. This input should be from $images(n) = 0$ up to a maximum value of the number of plotted time-points. It has a vector length equal to $n.graph$:

$$in.images\{n\} = 0 \ldots points(1)$$

in.imagetype (={1,1..}): This is the *type* of transverse image plotted. If $imagetype = 1$, a perspective surface plot is output, otherwise if $imagetype = 2$ a gray plot with colours is output, or if if $imagetype = 3$ contour plot with 10 equally spaced contours is generated. This has a vector length equal to $n.graph$.

$$in.imagetype\{n\} = 1, 2, 3$$

in.transverse (={0,0..}): This is the number of 2D, transverse o-x images plotted as discrete time slices. Only valid if $dimension > 2$. Note that, if present, the y,z-coordinates are set to their central values, when plotting the transverse images. This input should be from $transverse(n) = 0$ up to a maximum value of the number of plotted time-points. It has a vector length equal to $n.graph$:

$$in.transverse\{n\} = 0 \ldots points(1)$$

in.headers (={1,1,..}): This is a Boolean variable with value *true or* 1 if graphs require headers giving the simulation name, and *false* or 0 with no headers. It is useful to include headings on graphs in preliminary stages, while they may not be needed in a published final result.

$$in.headers\{n\} = 0, 1$$

in.pdimension (={4,4,..}): This is the maximum plot dimension for each graphed quantity. The purpose is eliminate unwanted graphs. For example, it is useful to reduce the maximum dimension when averaging in space. Higher dimensional graphs are not needed, as the data is duplicated. Averaging can be useful for checking conservation laws, or for averaging over homogeneous data to reduce sampling errors.

$$in.pdimension\{n\} = 1..4$$

in.xlabels (={'t','x','y','z'}): Labels for the graph axis independent variable labels, vector length of *dimension. Note, these are typeset in Latex mathematics mode!*

$$in.xlabels = \{xlabels(1), \ldots xlabels\,(dimension)\}$$

in.klabels (={'\omega','k_x','k_y','k_z'}): Labels for the graph axis Fourier transform labels, vector length of *dimension. Note, these are typeset in Latex mathematics mode!*

$$in.klabels = \{klabels(1), \ldots olabels\,(dimension)\}$$

4.5.3 Graphics projections

If there is a spatial lattice, the graphics program automatically generates several graphs for each observable, depending on space dimension. The maximum dimension that is plotted as set by *pdimension*. In the plots, the lattice is projected down to successively lower dimensions.

For each observable, the projection sequence is as follows:

- If **dimension**=**4**, a central z point $nz = 1 + floor(in.points(4)/2)$ is picked. For example, with 35 points, this gives the central point, $nz = 18$.
- This gives a three dimensional space-time lattice, which is treated the same as if **dimension** = **3**.
- If **images** are specified, two dimensional $x-y$ plots are generated at equally spaced time intervals. If there is only one image, it is at the last time-point. Different plot-types are used depending on the setting of **imagetype**.
- A central y point $ny = 1 + floor(in.points(3)/2)$ is picked. This gives a two dimensional space-time lattice, which is treated the same as if **dimension** = **2**. If **transverse** is specified, one dimensional x plots are generated at equally spaced time intervals, as before.
- A central x point $nx = 1 + floor(in.points(2)/2)$ is picked. This gives a one dimensional time lattice, which is treated the same as if **dimension** = **1**.
- Plots of observable vs time are obtained, including sampling errors and error bars. If comparison graphs are specified using *compare* functions, they are plotted also, using a dotted line. A difference graph is also plotted when there is a comparison.

4.6 Averages and integrals

4.6.1 Averages

Lattice averages can allow one to extract stochastic results with reduced sampling errors. An average over the lattice is carried out using the *xave* function, which is defined as follows:

xave(o,dx,r) This function takes a scalar observable $o = [1, lattice]$, defined on the xSPDE lattice, and returns a space average with dimension $[1, lattice]$. The input is an observable o, and an optional lattices structure and vector switch dx. If $dx(j) > 0$ an average is taken over dimension j. Dimensions are labelled from $j = 1, \ldots 4$ as elsewhere. Time averages are ignored at present. Averages are returned at all lattice locations. To average over samples and all space dimensions, just use ***xave(o)***.

Higher dimensional graphs of lattice averages are generally not useful, as they are simply flat. The xSPDE program allows the user to remove unwanted higher dimensional graphs of average variables. This is achieved by setting the corresponding element of *pdimension* to the highest dimension required, which of course depends on which dimensions are averaged.

For example, to average over the entire space lattice and indicate that only time-dependent graphs are required, set $dx = in.dx$ and:

$$in.pdimension = 1$$

Note that *xave* does not perform any average over ensembles, although this is done elsewhere for results calculated in any of the *observe* functions.

4.6.2 Integrals

Integrals over the spatial lattice allow calculation of conserved or other global quantities. The code to take an integral over the lattice is carried out using the xSPDE *xint* function:

xint(o,dx,r) This function takes a scalar o, and returns a space integral over selected dimensions with vector measure dx. If $dx(j) > 0$ an integral is taken over dimension j. Dimensions are labelled from $j = 1, \ldots 4$ as in all xspde standards. Time integrals are ignored at present. Integrals are returned at all lattice locations. To integrate over an entire lattice, set $dx = r.dx$, otherwise set $dx(j) = r.dx(j)$ for selected dimensions j.

As with averages, the xSPDE program allows the user to remove unwanted higher dimensional graphs when the integrated variable is used as an observable. For example, in a four dimensional simulation with integrals taken over the y and z coordinates, only t and x dependent graphs are required. Hence, set $dx = [0, 0, r.dx(3), r.dx(4)]$, and:

$$in.pdimension = 2$$

If momentum-space integrals are needed, use the transform switch to make sure that the field is Fourier transformed, and input $r.dk$ instead of $r.dx$. Note that *xint* returns a lattice observable, as required when used in the *observe* function. If the integral is used in another function, note that it returns a matrix of dimension $[1, lattice]$.

4.7 Frequently asked questions

Answers to some frequent questions, and reminders of points in this chapter are:

- Can you average other stochastic quantities apart from the field?
  - Yes: just specify this using the user function **in.observe**.
- Can you have functions of the current time and space coordinate?

  - Yes: xSPDE functions support this using the structure r, as $r.t$,$r.x$,$r.y$,$r.z$.

- Can you have several variables?
  - Yes, input this using **in.fields** $> \mathbf{1}$.
- Are higher dimensional differential equations possible?
  - Yes, this requires setting **in.dimension** $> \mathbf{1}$.
- Can you have spatial partial derivatives?
  - Yes, provided they are linear in the fields; these are obtainable using the function **in.linear**.
- Can you delete the graph heading?
  - Yes, this is turned off if you set **in.headers** $= \mathbf{0}$.
- Why are there two lines in the graphs sometimes?
  - These are the one standard deviation sampling error limits, generated when **in.ensembles**$(\mathbf{2}, \mathbf{3}) > \mathbf{1}$.
- Why is there just one line in some graphs, with no sampling errors indicated?
  - You need ensembles(2) or (3) for this; see previous question.
- What are the error bars for?
  - These are the estimated maximum errors due to finite step-sizes in time.

Details of how to use the parameters and functions, and a tutorial are given in the following chapters.

5 Tutorial

This chapter is a a tutorial in xSPDE functionality, giving a number of examples, and exercises. Not all the graphs generated by the scripts are included here, for space reasons. One can obtain many more graphs if desired, by generating more observables.

Vertical bars in the graphs are the step-size errors in time, calculated from setting $in.errorchecks = 2$. These are automatically omitted when the relative errors are too small to be visible. In most cases, the default ranges and step-sizes are used to keep things simple. One can improve this accuracy by using more points, as shown in the first example, or by using more steps per point.

Upper and lower solid lines are due to sampling errors. This occurs where there is statistical noise, and requires a finite number of serial *(in.ensembles(2))* or parallel *(in.ensembles(3)) ensembles* to calculate it. One can improve this by using more ensembles. Sub-ensemble averaging with multiple sub-ensembles is used to improve the accuracy of error estimates.

There are preset preferences for all the input parameters except the derivative function, which defines the equation that is simulated. Each example in the tutorial has exercises, which are very simple. However, they help to understand xSPDE conventions, and it is recommended to try them.

All the exercises, and some bonus examples, are given in the xSPDE Examples folder.

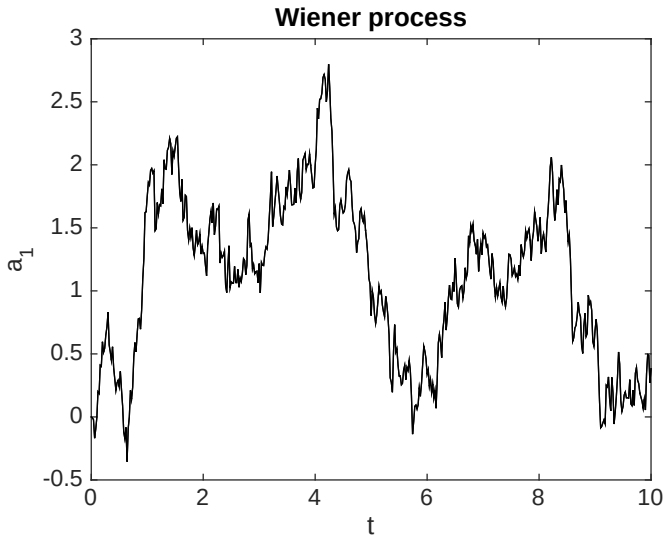


Figure 5.1: Increasing the Wiener process resolution and adding a header.

5.1 Wiener process

Try increasing the time resolution and adding a heading to the random walk example in Chapter 3. This requires specifying the number of points using *in.points*. To name the simulation, use *in.name*, which is stored with your simulation data. The default option is to add this heading to each graph. If no header is wanted, type $in.headers = 0$.

```
in.name = 'Wiener process';
 in.points = 501;
in.da = @(~,z,~) z;
 xspde(in);
```

To run the xSPDE program after adding these inputs, just press the *Run* icon on the Matlab editor bar. This will run the xSPDE program, with default parameters where they are not specified in the inputs. You will see the following figure:

Exercise:

Add 100 samples and 100 serial ensemble trajectories. Does the mean equal zero within the sampling error bars?

5.2 Harmonic oscillator

The next example is the stochastic harmonic oscillator with the initial condition that

$$a(0) = 1 + w\,,$$

where

$$\left\langle w^2 \right\rangle = 1$$

and the differential equation:

$$\dot{a} = ia + \zeta\,.$$

5.2.1 Initial conditions and derivative

First, make sure you type 'clear' to clear the previous example. This is good practice for all the examples. The following parameters are needed to specify the harmonic oscillator with noise. By specifying return values in square brackets, the data is made available in the user data space:

```
in.initial = @(v,~) 1+v;
in.ensembles = [20,20];
in.da = @(a,z,~) i*a + z;
in.olabels = {'<a_1>'};
xspde(in);
```

Here '~' indicates an unused input to a function, while 'i' is the Matlab codes for the unit imaginary number, i. The following graph is produced:

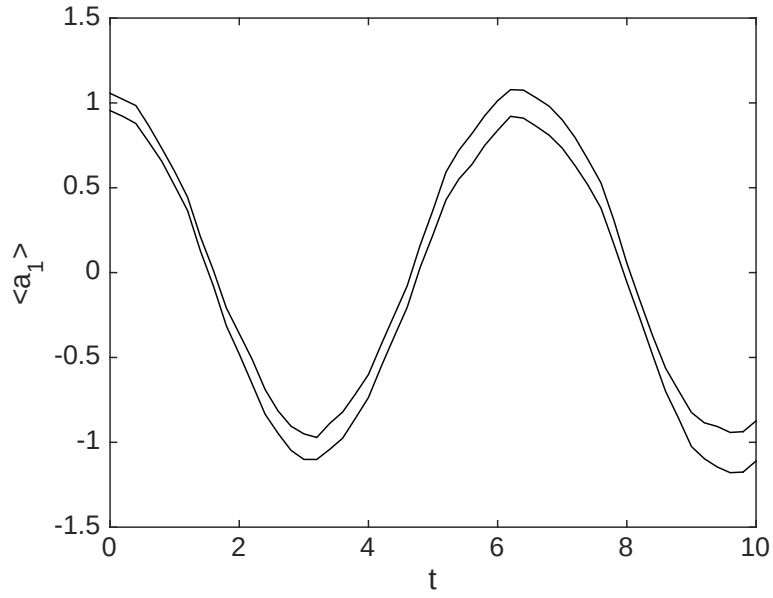


Figure 5.2: Simple harmonic oscillator amplitude

The plotted error-bars are suppressed, as they are too small to see, nor is there any header, since none was specified. The **xsim** program reports the following error summary, using the default number of points (51):

- Max sampling error = 1.193890e-01
- Max step error = 1.270917e-02

This is an approximate upper bound on the overall integration error of the specified observable. It is calculated from comparing two solutions. In this case, the default estimates are obtained by comparing a coarse and fine step calculation at half the specified step-size. This is used to extrapolate to zero step-size. The difference between the fine result and the extrapolated result gives the error estimate.

5.2.2 Comparisons with exact results

The stochastic equation has the mean solution:

$$\begin{aligned} \langle a(t) \rangle &= e^{it} \\ &= \cos(t) + i \sin(t) \end{aligned}$$

To compare the calculated solution with this exact result, just tell the graphics program that you want a comparison, by editing the project file, and adding a comparison function.

This example uses the previous inputs, together with the comparison function itself (*in.compare*). All functions and data relating to observables are cell arrays, hence the

curly brackets: *gr.compare*{1} is the first element of an array of comparison functions that might be needed if there are many observables.

```
in.compare{1} = @(t,~) cos(t);
xspde(in);
```

With this input, xgraph gives the difference in the comparison as:

- Maximum comparison differences = 1.950535e-01

The actual error in this case is smaller than the error estimated using the sampling error estimates. However, the error-bars are very small. This is because in this case, the specified fine step-size is small enough to give excellent convergence.

Comparison graphs are also produced, including one of the relative errors:

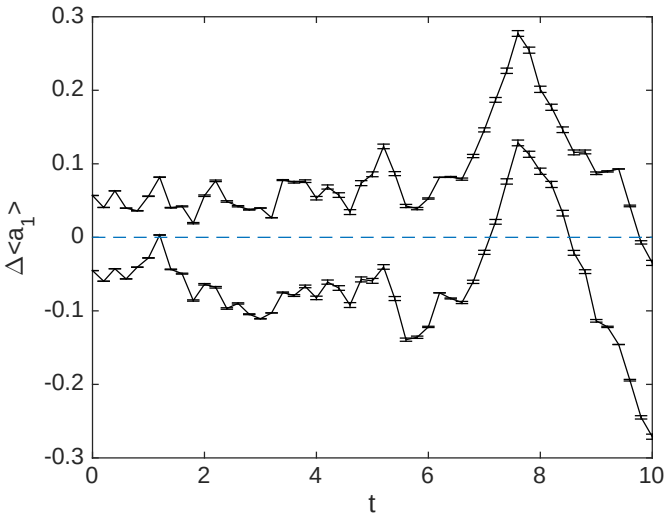


Figure 5.3: Simple harmonic oscillator comparison graph: exact vs computed, with error-bars.

The reported summary data is consistent with the graphs, as expected. Note that one can obtain exactly the same result in the interaction picture, by using an imaginary linear coupling of i, and a derivative term of zero. The code then reports a maximum step-size error of around $\sim 10^{-15}$, equal to the limit of IEEE arithmetic.

Exercise:

Add a linear decay of $-a$ to the differential equation, and modify the exact solution to suit, then replot. Is it exactly as you expected?

5.3 Kubo oscillator

The next example is more interesting. It is the Kubo oscillator, an oscillator with a random frequency. In Stratonovich stochastic calculus, its equation is:

$$da = ia.dw$$

Given the initial condition that $a(0) = 1$, each trajectory has the solution:

$$a\left(t\right) = e^{iw(t)}$$

where

$$w(t) = \int_0^t dw$$

The corresponding mean value is different to the instantaneous trajectory, owing to dephasing:

$$\left\langle a\left(t\right)\right\rangle = e^{-\left\langle w^2(t)\right\rangle/2} = e^{-t/2}.$$

5.3.1 Kubo initial conditions and derivative

Here more parameters are needed. One real noise term is required per integration point, specified using *in.noises*. Next, the ensemble numbers are required. Here we use 100 vector-level trajectories, and 16 sets at a higher level. In these calculations, the mean amplitude is calculated, and compared against a comparison function.

```
function e = Kubo()
in.name = 'Kubo oscillator';
in.ensembles = [400,16,1];
in.initial = @(v,r) 1+0*v;
in.da = @(a,z,r) i*z.*a;
in.olabels = {'<a_1>'};
in.compare{1}= @(t,~) exp(-t/2);
e = xspde(in);
end
```

Kubo error results are reported as:

- Max sampling error = 1.043423e-02
- Max step error = 2.258936e-02

Note that these are generally consistent with the graphs below, as they should be.

Is the actual error always less than the reported maximum standard deviation? This is not always the case, for statistical reasons. The statistical estimates given are best estimates of the standard deviations of the plotted means. However, given a large enough

number of means at different times, some **must** fall outside the range of a unit standard deviation.

The different time points in the Kubo oscillator trajectories become uncorrelated after a time of order one. Hence an occasional excursion with an error of 2σ can occur. In other words, the expected maximum sampling error is a multiple of the standard deviation, which should therefore be treated with some caution as a guide to statistical errors.

We see evidence here the sampling errors often exceed the step-size errors, unless large sample numbers are used.

5.3.2 Kubo graphs

The Graphics program reports the following errors when making the comparisons:

- Max difference in 1 = 1.294696e-02

With this choice of algorithm and step-size, the results of a simulation run are plotted below.

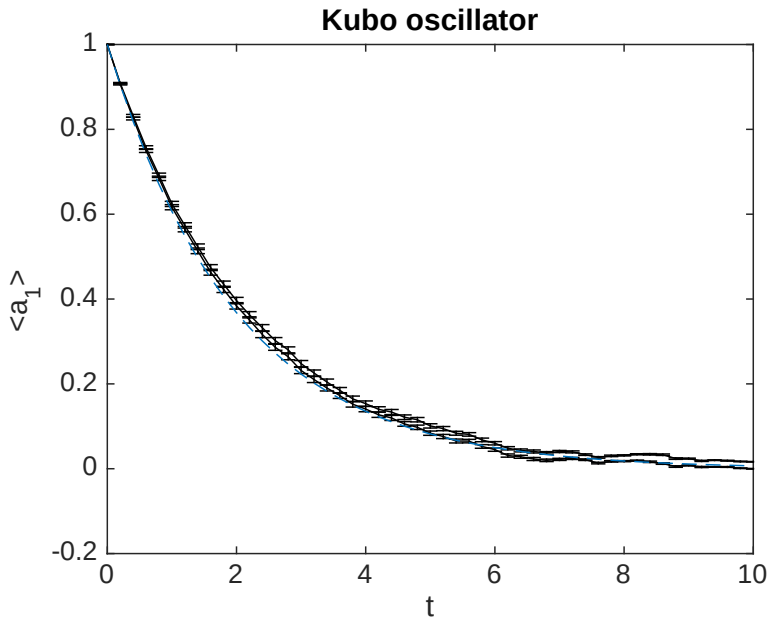


Figure 5.4: Kubo oscillator mean amplitude

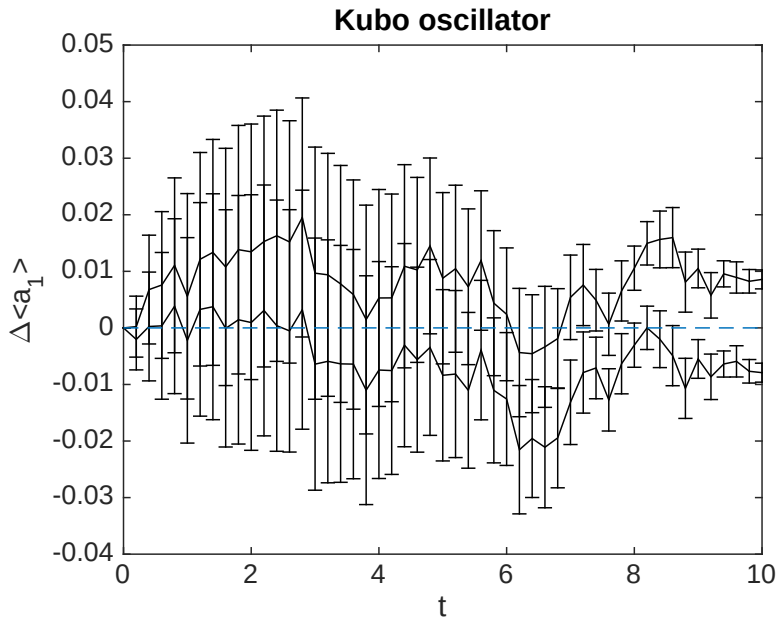


Figure 5.5: Kubo oscillator amplitude errors

There are some interesting features here. The two solid lines indicate the sampling error. The error bars indicate the step-size error. This affects both results, but is only visible in the error graphs, which have an expanded scale.

Exercise:

Add a detuning of ia to the differential equation, modify the exact solution to suit, then replot.

5.4 Soliton

The third example is the soliton equation for the nonlinear Schrödinger equation, with:

$$\frac{da}{dt} = \frac{i}{2}\left[\nabla^2 a - a\right] + ia\left|a\right|^2$$

Together with the initial condition that $a(0,x) = sech(x)$, this has an exact solution that doesn't change in time:

$$a(t,x) \quad = \quad sech(x)$$

The Fourier transform at $k = 0$ is simply:

$$\tilde{a}(t,0) \quad = \quad \frac{1}{\sqrt{2\pi}}\int sech(x)dx = \sqrt{\frac{\pi}{2}}$$

5.4.1 Soliton parameters and functions

The important parameters and functions in this case are:

```
function [e] = Soliton()
in.name = 'NLS soliton';
in.dimension = 2;
in.initial = @(v,r) sech(r.x);
in.da = @(a,~,r) i*a.*(conj(a).*a);
in.linear = @(D,r) 0.5*i*(D.x.^2-1.0);
in.olabels = {'a_1(x)'};
in.compare{1}= @(t,~) 1;
e = xspde(in);
end
```

The xspde program reports the following maximum errors:

- Max step error = 1.976729e-02

The output reflects the known analytic result.

5.4.2 Soliton graphs and errors

Graphs of results are given below.

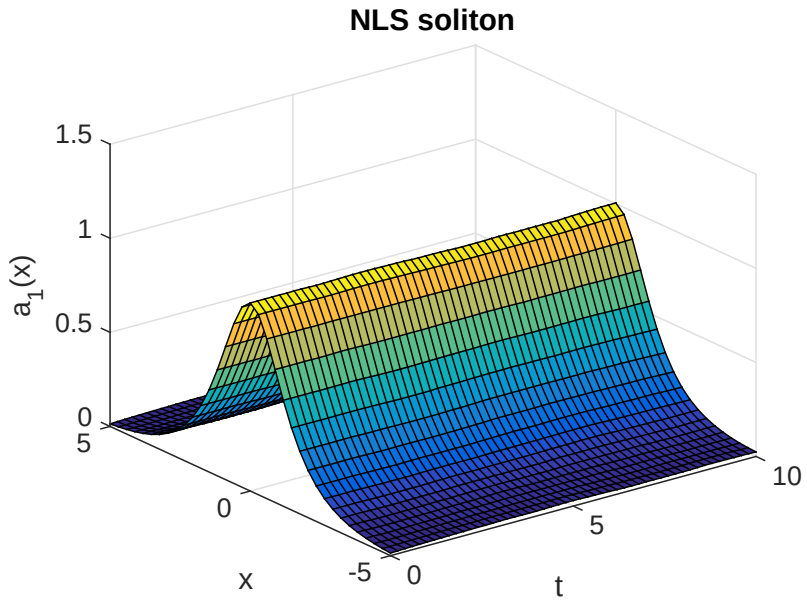


Figure 5.6: Soliton amplitude versus space and time

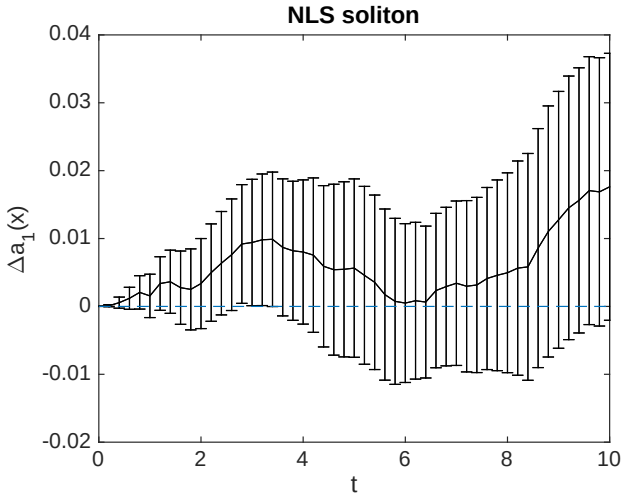


Figure 5.7: Soliton amplitude errors at center

The xgraph program reports that comparison errors are slightly less than the step error:

- Max difference in 1 = 1.761991e-02

This is not always the case, because the error checking does not check errors due to the lattice sizes. In general this needs to be carried out manually.

Exercise:

Add an additive complex noise of $0.01(dw_1 + idw_2)$ to the differential equation, then replot with an average over 1000 samples.

5.5 Gaussian with HDF5 files

The fifth example is free diffraction of a Gaussian wave-function in three dimensions, given by

$$\frac{da}{dt} = \frac{i}{2}\nabla^2 a$$

Together with the initial condition that $a(0,x) = exp(-|\mathbf{x}|^2/2)$, this has an exact solution for the diffracted intensity in either ordinary space or momentum space:

$$\begin{aligned} |a(t,\mathbf{x})|^2 &= \frac{1}{(1+t^2)^{3/2}} exp(-|\mathbf{x}|^2/\left(1+t^2\right)) \\ |\tilde{a}(t,\mathbf{k})|^2 &= exp(-|\mathbf{k}|^2) \end{aligned}$$

5.5.1 Gaussian inputs

A possible user set of parameters to simulate this is:

```
function [e] = Gaussian()
in.dimension = 4;
in.initial = @(v,r) exp(-0.5*(r.x.^2+r.y.^2+r.z.^2));
in.da = @(a,~,~) zeros(size(a));
in.linear = @(D,r) 1i*0.05*(D.x.^2+D.y.^2+D.z.^2);
in.observe = {@(a,~) a.*conj(a)};
in.olabels = {'|a(x)|^2'};
in.HDF5file = {'Gaussian.f5'};
in.images = 4;
in.imagetype = 1;
in.transverse = 2;
in.headers = 1;
in.compare{1} = @(t,~) [1+(t/10).^2].^(-3/2);
e = xsim(in);
e = xgraph('',in);
end
```

Here the program writes an HDF5 data file using *xsim*, and then reads it in with the stored file-name, using *xgraph*. The program reports the following maximum step-size errors, which in this case are negligible, as they are purely due to the interaction picture transformations:

- Max step error = 4.107825e-15

However, the finite spatial lattice size introduces errors in the on axis intensity, in coordinate space. This shows up in the comparisons:

- Max difference in 1 = 5.590272e-07

5.5.2 Gaussian graphs

With this choice of algorithm and step-size, the results of a simulation run are plotted below. The errors, of order 10^{-7}, are simply due to interference of diffracted waves caused by the periodic boundary conditions. This is sometimes called aliasing error. One can think of this physically as being a simulation of an infinite array or periodically repeated Gaussian inputs, which can diffract and interfere.

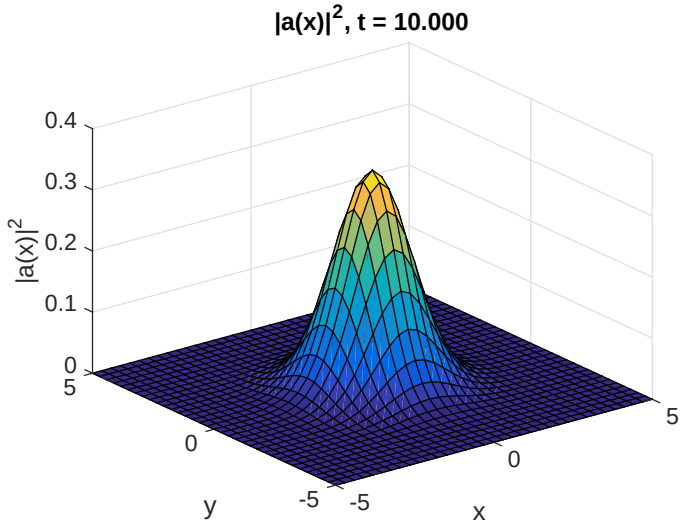


Figure 5.8: Image of transverse gaussian intensity at $t = 0$.

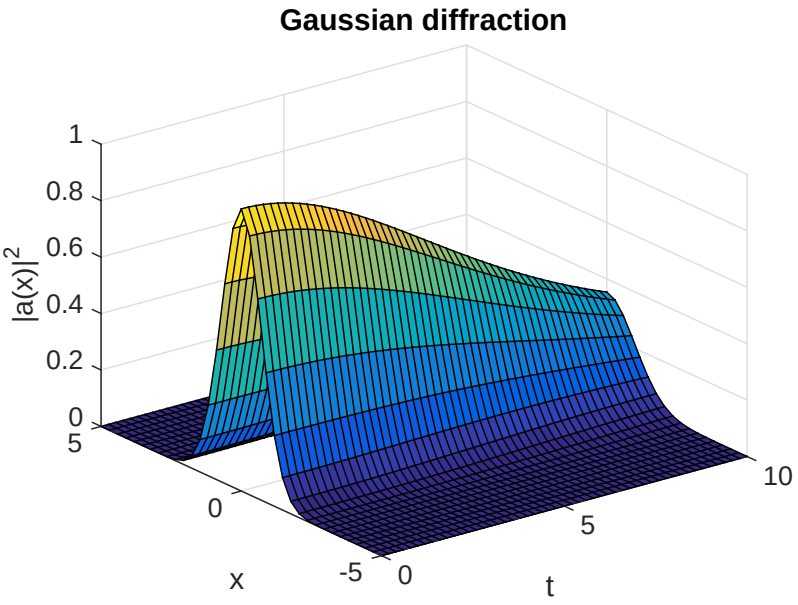


Figure 5.9: Gaussian intensity diffraction

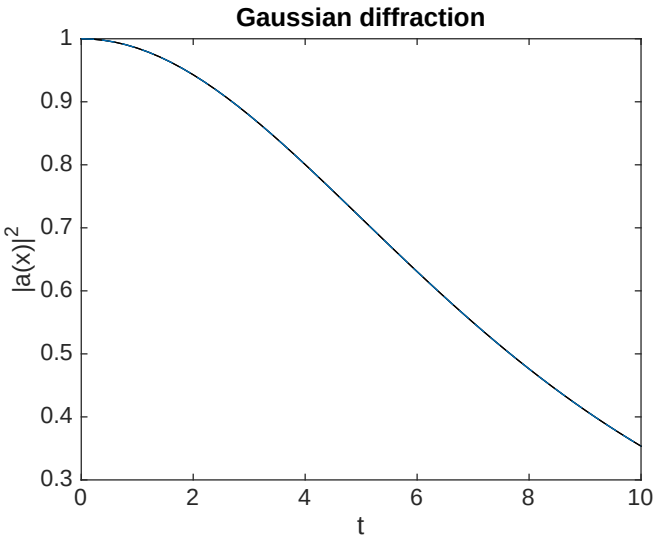


Figure 5.10: Gaussian intensity at $\boldsymbol{r} = 0$.

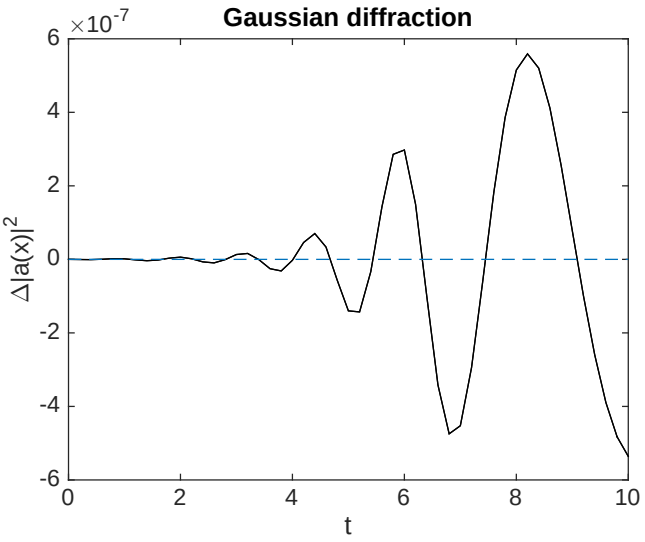


Figure 5.11: Gaussian, modulus-squared errors at $\boldsymbol{r} = 0$.

Exercise:

Add an additive complex noise of $0.01(dw_1 + idw_2)$ to the Gaussian differential equation, then replot with an average over 10 samples.

5.6 Planar noise

The fifth example is growth of thermal noise of a two-component complex field in a plane, given by the equation

$$\frac{d\boldsymbol{a}}{dt} = \frac{i}{2}\nabla^2\boldsymbol{a} + \boldsymbol{\zeta}(t,x)$$

where $\boldsymbol{\zeta}$ is a delta-correlated complex noise vector field:

$$\zeta_j(t,\mathbf{x}) = \left[\zeta_j^{re}(t,\mathbf{x}) + i\zeta_j^{im}(t,\mathbf{x})\right]/\sqrt{2},$$

with the initial condition that the initial noise is delta-correlated in position space

$$a(0,\mathbf{x}) = \boldsymbol{\zeta}^{(in)}(\boldsymbol{x})$$

where:

$$\boldsymbol{\zeta}^{(in)}(\boldsymbol{x}) = \left[\boldsymbol{\zeta}^{re(in)}(\mathbf{x}) + i\boldsymbol{\zeta}^{im(in)}(\mathbf{x})\right]/\sqrt{2}$$

This has an exact solution for the noise intensity in either ordinary space or momentum space:

$$\begin{aligned}
\left\langle \left|a_j\left(t,\mathbf{x}\right)\right|^2\right\rangle &= (1+t)/\Delta V \\
\left\langle \left|\tilde{a}_j\left(t,\mathbf{k}\right)\right|^2\right\rangle &= (1+t)/\Delta V_k \\
\left\langle \tilde{a}_1\left(t,\mathbf{k}\right)\tilde{a}_2^*\left(t,\mathbf{k}\right)\right\rangle &= 0
\end{aligned}$$

Here, the noise is delta-correlated, and ΔV, ΔV_k are the cartesian space and momentum space lattice cell volumes respectively. Suppose that $N = N_x N_y$ is the total number of spatial points, and $V = R_x R_y$, where there are $N_{x(y)}$ points in the x(y)-direction, with a total range of $R_{x(y)}$. Then, $\Delta x = R_x/N_x$,$\Delta k_x = 2\pi/R_x$, so that:

$$\begin{aligned}
\Delta V &= \Delta x \Delta y = \frac{V}{N} \\
\Delta V_k &= \Delta k_x \Delta k_y = \frac{(2\pi)^2}{V}.
\end{aligned}$$

In the simulations, two planar noise fields are propagated, one using noise generated in position space, the other with noise generated in momentum space. This example shows that, provided no filters are applied, both types of noise are identical in their effects. However, momentum space noise requires an N-dimensional inverse FFT before being added, which is slower, so this method is not recommended unless needed.

5.6.1 Planar inputs

```
function [e] = Planar()
in.name = 'Planar noise growth';
in.dimension = 3;
in.fields = 2;
in.ranges = [1,5,5];
in.steps = 2;
in.noises = [2,2];
in.ensembles = [10,4,4];
in.initial = @Initial;
in.da = @Da;
in.linear = @Linear;
in.observe{1} = @(a,~) a(1,:).*conj(a(1,:));
in.observe{2} = @(a,~) xave(a(1,:).*conj(a(1,:)));
in.observe{3} = @(a,~) xave(a(2,:).*conj(a(2,:)));
in.observe{4} = @(a,~) xave(a(1,:).*conj(a(2,:)));
in.transforms = {[0,0,0],[0,0,0],[0,1,1],[0,1,1]};
in.olabels{1} = '<|a_1(x)|^2>';
in.olabels{2} = '<<|a_1(x)|^2>>';
in.olabels{3} = '<<|a_2(k)|^2>>';
in.olabels{4} = '<<a_1(k)a^*_2(k)>>';
in.compare{1} = @(t,in) [1+t]/in.dV;
in.compare{2} = @(t,in) [1+t]/in.dV;
in.compare{3} = @(t,in) [1+t]/in.dK;
in.compare{4} = @(t,in) 0;
in.images = [4,2,0,0];
in.transverse = [2,2,0,0];
in.pdimension = [4,1,1,1];
e = xspde(in);
end

function a0 = Initial(v,r)
a0(1,:)  = (v(1,:)+1i*v(2,:))/sqrt(2);
a0(2,:)  = (v(3,:)+1i*v(4,:))/sqrt(2);
end

function da = Da(a,z,r)
da(1,:)  = (xi(1,:)+1i*xi(2,:))/sqrt(2);
da(2,:)  = (xi(3,:)+1i*xi(4,:))/sqrt(2);
end

function L = Linear(D,r)
lap = D.x.^2+D.y.^2;
L(1,:)  = 1i*0.5*lap(:);
L(2,:)  = 1i*0.5*lap(:);
end
```

5.6.2 Planar graphs

With this choice of algorithm and step-size, the results are plotted below.

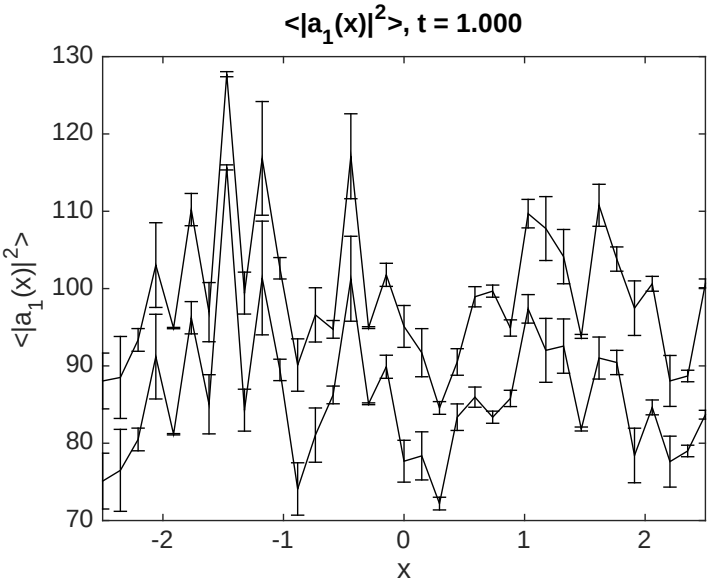


Figure 5.12: Planar noise intensity as a transverse slice in the $t = 1, y = 0$ plane. The relatively large sampling error is because there are not many samples.

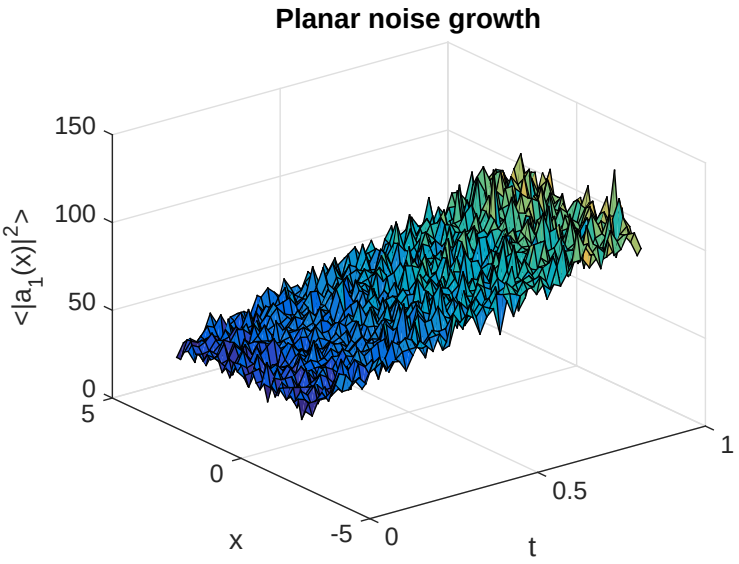


Figure 5.13: Growth in noise intensity with time vs. x, at $y = 0$

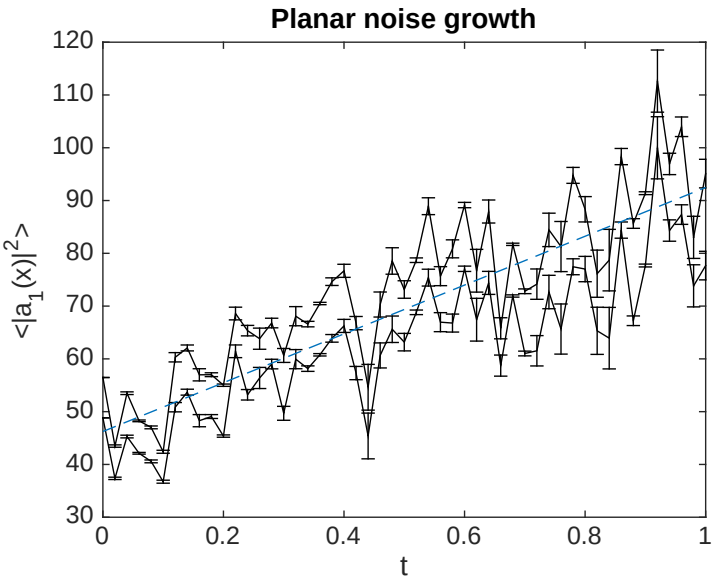


Figure 5.14: Growth in planar noise intensity at $x = y = 0$, vs. exact results

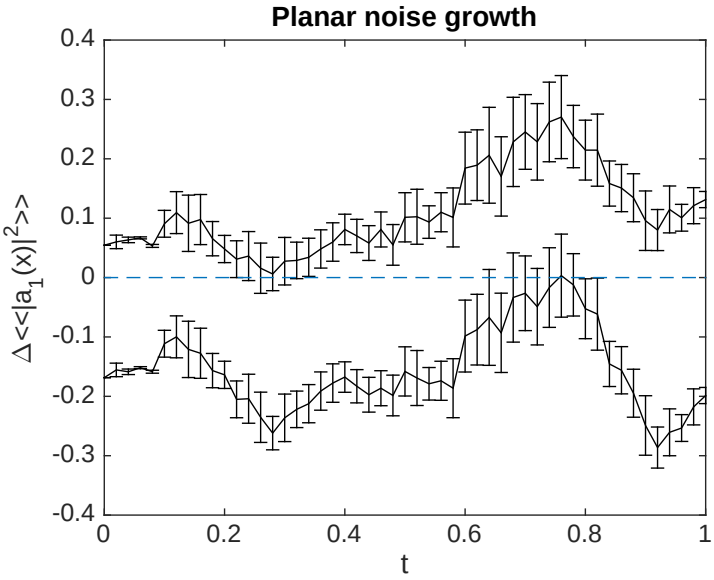


Figure 5.15: Errors in planar noise intensity at $x = y = 0$, vs. exact results. These results are averaged across the plane, as well as being ensemble averaged.

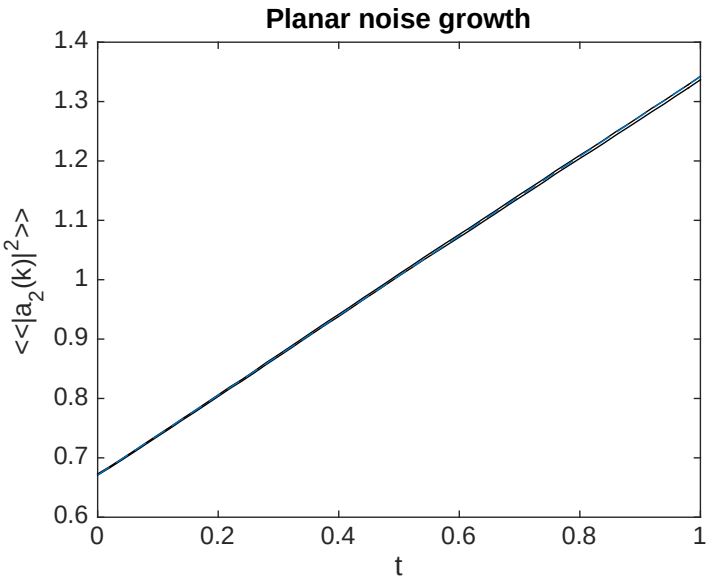


Figure 5.16: Growth in planar noise intensity in momentum space, for the second field, at $k_x = k_y = 0$.

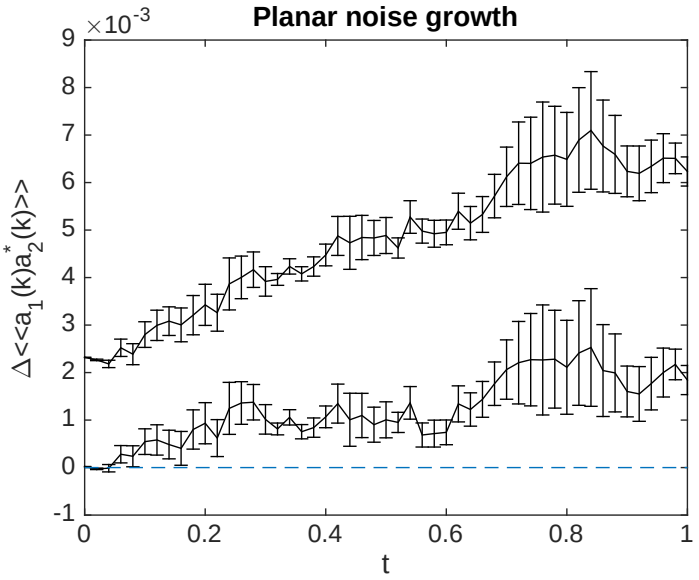


Figure 5.17: Lattice averaged errors in cross-correlations in momentum space, vs. exact results.

Exercise:

Add a decay rate of $-a$ to the Planar differential equation, then replot.

5.7 Extensible simulations

Next, an extensible simulation: first a noisy absorber, then a noisy amplifier. The second part has a different differential equation, and larger graphical scales.

This is handled with the extensibility feature of xSPDE. Just enter a sequence of inputs, in the form *{in1,in2,in3..}* with a corresponding sequence of graphs, *{g1,g2,g3..}*. Here, the first equation is:

$$\frac{da}{dt} = -a + \zeta_1(t) + i\zeta_2(t)$$

with an initial condition of $a = 1$. The mean intensity is constant:

$$\left\langle |a(t)|^2 \right\rangle = 1\,.$$

5.7.1 Input file

The full input file is given below.

```
function [e] = Gain()
in.name = 'Loss with noise';
in.ranges = 4;
in.noises = [2,0];
in.ensembles = [100,16,1];
in.initial = @(v,~) (v(1,:)+1i*v(2,:))/sqrt(2);
in.da = @(a,z,r) -a + z(1,:)+1i*z(2,:);
in.observe{1} = @(a,~,~) a.*conj(a);
in.olabels = {'|a|^2'};
in.compare = {@(t,~) 1+0*t};
in2 = in;
in2.steps = 4;
in2.origin = in.ranges;
in2.name = 'Gain with noise';
in2.da = @(a,z,r) a + z(1,:)+1i*z(2,:);
in2.compare = {@(t,~) 2*exp(2*(t-4))-1};
e = xspde({in,in2});
end
```

Note that the code defines $in2 = in$ and $gr2 = gr$ before making any changes, so that only a few additional inputs are needed. The number of *steps* is increased to improve the accuracy of the second integration, and the second time origin is chosen so that it starts from the time the first simulation is completed.

Results are graphed below.

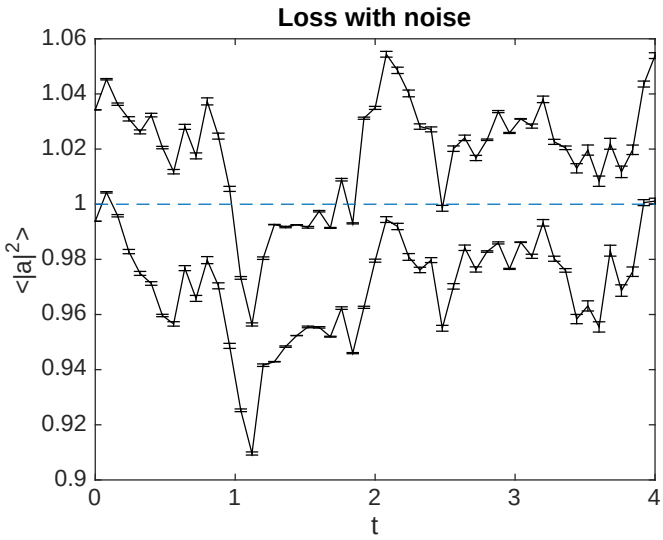


Figure 5.18: Absorber intensity

Comparison graphs are also produced for the relative errors. In the graph given here,

5.7.2 Extended simulations

The second differential equation has an initial condition corresponding to the solution of the first equation at $t = 4$, and the derivative:

$$\frac{da}{dt} = a + \zeta_1(t) + i\zeta_2(t)$$

The mean intensity grows exponentially:

$$\left\langle |a|^2 \right\rangle = 1\,.$$

$$\left\langle |a(t)|^2 \right\rangle = 2e^{2(t-4)} - 1$$

where

$$w(t) = \int_0^t \zeta(t')dt'$$

To compare the calculated solution with this exact result, there are two Compares functions in the project file. Note that the code defines $in2 = in$ and $g = g2$ before making any changes, so that only a few additional inputs are needed. The number of *steps* is increased to improve the accuracy of the second integration. The time axis in the second graph has the origin reset to zero.

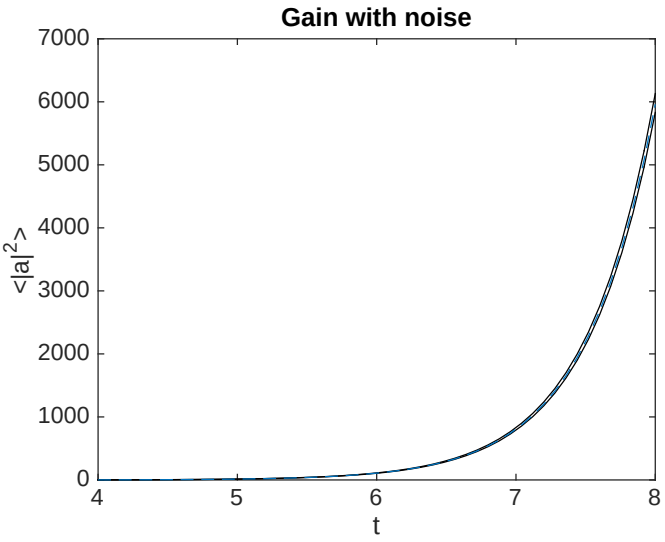


Figure 5.19: Noisy amplifier intensity

Comparison graphs of the relative errors are also produced here as well.

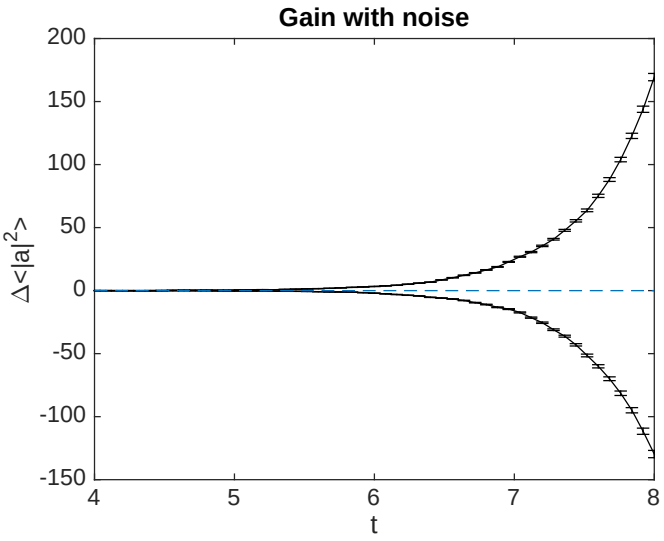


Figure 5.20: Noisy amplifier intensity errors, showing how the sampling errors increase in time.

Exercise:

Reverse the order of gain and loss.

5.8 Characteristic

The next example is the characteristic equation for a traveling wave at constant velocity,

$$\frac{da}{dt} + \frac{da}{dx} = 0$$

Together with the initial condition that $a(0,x) = sech(2x+5)$, this has an exact solution that propagates at a constant velocity:

$$a(t,x) \quad = \quad sech(2(x-t)+5)$$

The time evolution at $x = 0$ is simply:

$$a(t,0) \quad = \quad sech(2(t-5/2))$$

5.8.1 Characteristic inputs

The important parameters and functions in this case are:

```
function [e] = Characteristic()
in.name = 'Characteristic'
in.dimension = 2;
in.initial = @(v,r) sech(2.*(r.x+2.5));
in.da = @(a,z,r) 0*a;
in.linear = @(D,r) -D.x;
in.olabels = {'a_1(x)'};
in.compare = {@(t,in) sech(2.*(t-2.5))};
e = xspde(in);
end
```

The simulation program reports the following maximum errors:

- Max step error = 5.773160e-15

This is slightly misleading, since while the interaction picture is essentially exact, it is solving a finite lattice problem exactly. The transverse lattice discretization does introduce errors of course, and these are seen in the comparisons with the exact results:

- Maximum comparison differences = 7.581817e-03

Graphs of results are given below.

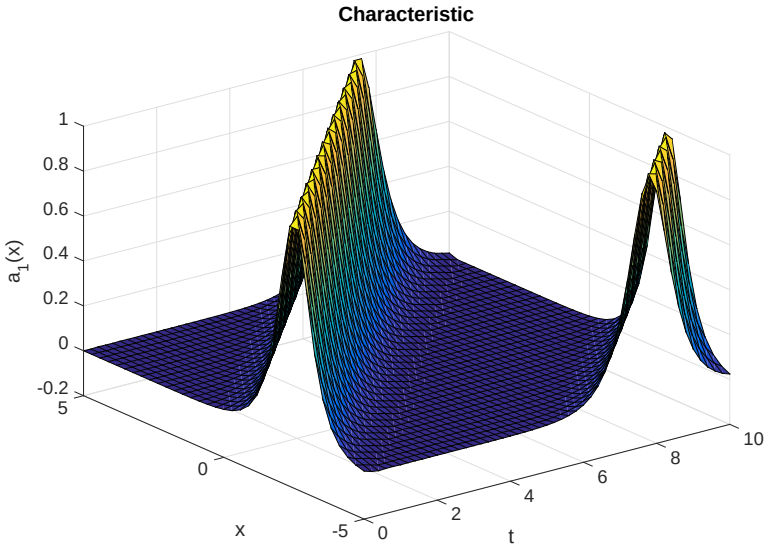


Figure 5.21: Characteristic traveling wave versus space and time

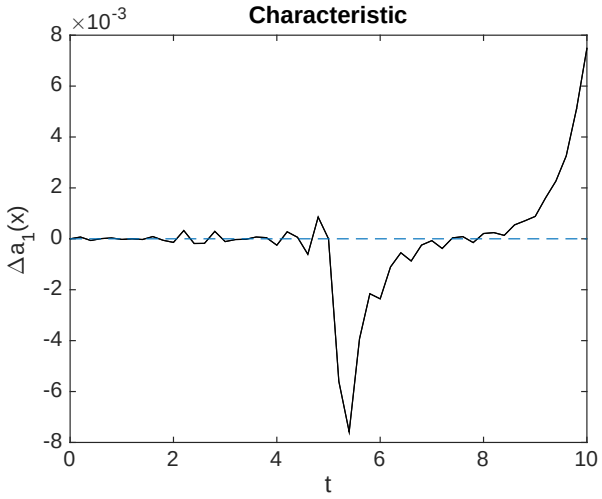


Figure 5.22: Characteristic errors at center

Exercise:

Recalculate with the opposite velocity, and a new exact solution.

5.9 Equilibrium

We now move on to frequency space simulations. The equation is the same as the earlier loss equation i.e.

$$\frac{da}{dt} = -a + \zeta(t)$$

where $\zeta(t) = \zeta_1(t) + i\zeta_2(t)$, with an initial condition of $a = (w_1 + iw_2)/\sqrt{2}$. For sufficiently long time-intervals, the solution is given by:

$$\tilde{a}\,(\omega) = \frac{\tilde{\zeta}(\omega)}{1 - i\omega}$$

The expectation value of the noise Fourier transform modulus squared, in the large T limit, is therefore:

$$\begin{aligned} \left\langle |\tilde{a}(\omega)|^2 \right\rangle &= \frac{1}{2\pi\,(1+\omega^2)} \int\int e^{i\omega(t-t')} \left\langle \zeta(t)\zeta^*(t') \right\rangle dtdt' . \\ &= \frac{T}{\pi\,(1+\omega^2)} \end{aligned}$$

5.9.1 Program inputs

The full input file is given below.

```
function e = Equilibrium()
in.name = 'Equilibrium spectrum';
in.points = 640;
in.ranges = 100;
in.noises = [2,0];
in.ensembles = [1000,10,1];
in.initial = @(v,~) (v(1,:)+1i*v(2,:))/sqrt(2);
in.da = @(a,z,r) -a + z(1,:)+1i*xi(2,:);
in.observe{1} =@(a,~) a.*conj(a);
in.observe{2} =@(a,~) a.*conj(a);
in.transforms ={0,1};
in.olabels = {'|a(t)|^2','|a(w)|^2'};
in.compare = {@(t,~) 1.+0*t, @(w,~)100./(pi*(1+w.^2))};
e = xspde(in);
end
```

Results are graphed below. The calculated spectrum is indistinguishable from the exact result.

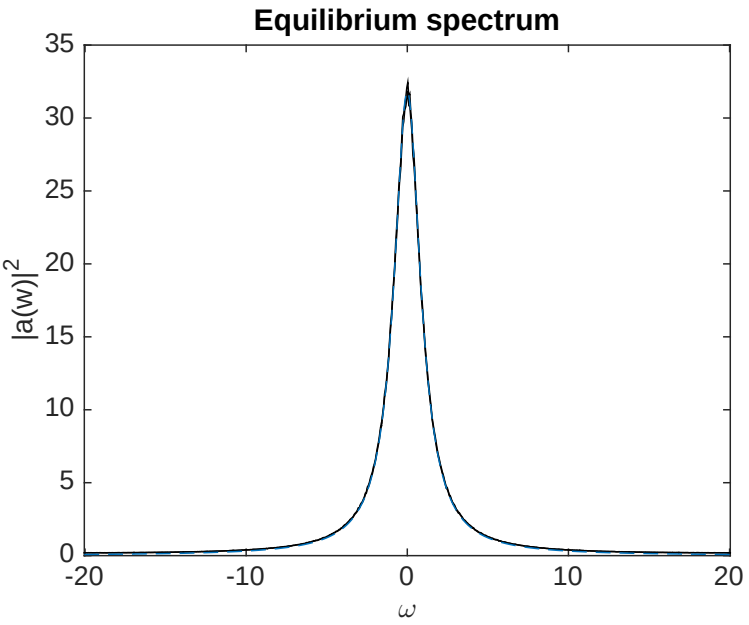


Figure 5.23: Equilibrium spectral intensity

The xsim program reports the following error summary:

- Max step error =5.856892e-02
- Max sampling error = 4.234763e-01
- Maximum comparison differences = 6.194415e-01

Here, the comparison differences indicate that the maximum error reported is actually about 1.5 standard deviations of the maximum sampling error. Given the large number of data points, this is a reasonable result.

Exercise:

Add a second field coupled to the first, so that:

$$\begin{aligned} \frac{da}{dt} &= -a + \zeta(t) \\ \frac{db}{dt} &= -b + a \end{aligned}$$

Compare the two spectra, and calculate what the second one should look like.

6 Logic and data

The simulation program logic is straightforward. It is a very compact function called **xspde**. This calls **xsim**, for the simulation, then **xgraph** for the graphics. Most of the work is done by other specialized functions. Input parameters come from an **input** array, output is saved either in a **data** array, or else in a specified file. When completed, timing and error results are printed.

6.1 How it works

To summarize the previous chapters, xSPDE will solve stochastic partial different equations for a vector field $\mathbf{a}(t, \boldsymbol{x})$ and vector noise $\boldsymbol{\zeta}(t, \boldsymbol{x})$, of form:

$$\frac{\partial}{\partial t}\mathbf{a}(t, \boldsymbol{x}) = \mathbf{A}\left[\mathbf{a}\right] + \underline{\mathbf{B}}\left[\mathbf{a}\right] \cdot \boldsymbol{\zeta}(t, \boldsymbol{x}) + \underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right] \cdot \mathbf{a} \tag{6.1}$$

It can also solve ordinary stochastic equations, or partial differential equations without noise. Extensive error checking outputs are available. Both initial stochastic conditions and noise can have nonlocal spatial filters applied. All inputs are entered as part of an object-oriented structure. This includes the functions used to specify the equations and the quantities to average. The outputs can be either stored, or graphed interactively. xSPDE includes built-in multidimensional graphics tools.

6.1.1 Input and data arrays

To explain xSPDE in full detail,

- Simulation inputs are stored in the **input** cell array.
- This describes a *sequence* of simulations, so that **input**=**{in1,in2,...}**.
- Each structure **in** describes a simulation, whose output is the input of the next.
- The main function is called using **data** =**xspde**(**input**).
- Averages are recorded sequentially in the **data** cell array.
- Raw trajectory data is stored in the **raw** cell array if required.

The sequence **input** has a number of individual simulation objects **in**. Each includes parameters that specify the simulation, with functions that give the equations and observables. If there is only one simulation, just one individual specification **in** is needed. In addition, xSPDE generates graphs with its own graphics program.

6.1.2 Customization options

There are a wide range of customization options available for those who wish to have the very own xSPDE version.

Customization options include functions the allow user definition of:

inputs

interfaces

stochastic equations

mean observables

linear propagators

coordinate grids

noise correlations

integration methods

There are four internal options for stochastic integration methods, but arbitrary user specification is also possible.

The program will print out a record of its progress, then generate the specified graphs.

6.2 Stochastic flowchart

The main program logic is nearly self-explanatory. It has four functions and two main arrays that store results.

There are also two important computational routines behind the scenes, which need to be kept in mind. These are **da**, which is short for difference in a. This is completely user specified, and gives a local step in time. The next workhorse routine is **xprop**. This is not a beefy Rugby forward, but calculates spatial propagation.

The logical order is as follows:

System functions

xsim decides the overall workflow, and parallel operation at a high level. Here, *in.ensembles(3)* is used to specify parallel integration, with a *parfor* loop. The random seeds include data from the loop index to make sure the noise is independent for each ensemble member, including parallel ensembles.

xlattice creates a space-time lattice from the input data, which is a data-structure. This also initializes the actual ***data*** array for averaging purposes. Next, a loop is initiated over an ensemble of fields for checking and ensemble averaging. The calculations inside the loop can all be carried our in parallel, if necessary. These internal steps are actually relatively simple.

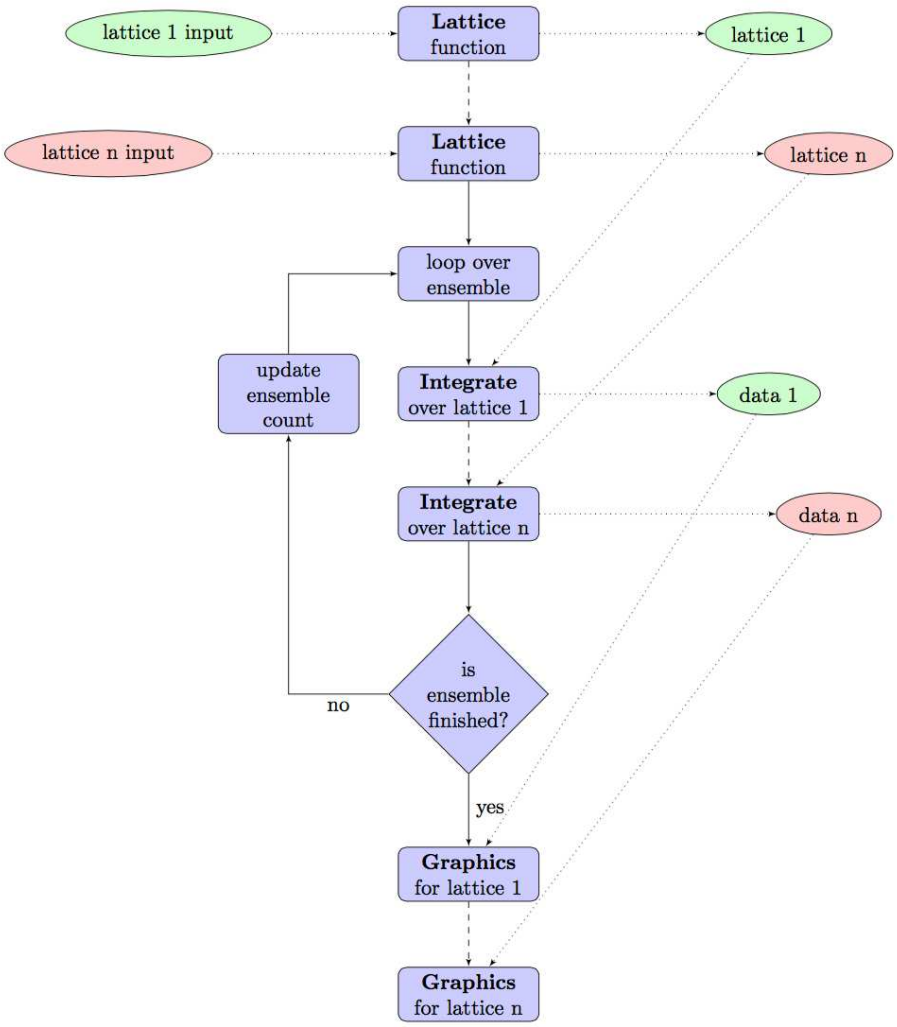


Figure 6.1: xSPDE flowchart, showing the data, lattice and field processing.

xinpreferences is called by xlattice to set the defaults that are not already entered.

xensemble repeats each stochastic path for the check/ensemble loop. It is important to notice that the random seed is reset at the start of each ensemble loop. The seed has a unique value that is different for each ensemble member. Note that for successive simulations that are **not** stored in the same data array, the seed should ideally be manually chosen differently for inputs to successive integration blocks, in order to guarantee independent noise sequences. The check variable can be set to $errorchecks = 1, 2$. This is the total number of integrations carried out. The integration is executed once with $errorchecks = 1$. With $errorchecks = 2$, there are two integrations, using half the step-size the second time. This takes three times as long overall. The matrices used to define the interaction picture transformations are stored for each check loop, as they vary with step-size.

xpath propagates the field ***a*** over a path in time. There are *in.steps* time-steps for each point stored in time, to allow for greater accuracy without excessive data storage, where needed. This integrates the equations for a predetermined time duration. Note that the random seed has the same value for **both** the check loops. This is because the same number of random variates must be generated in the same order to allow accurate extrapolation. The two loops must use the same random numbers, or else the check is not accurate. For random numbers generated during the integration, the coarse step will add two fine step random noises together, to achieve the goal of identical noise behavior. Results of any required averages, variances and checks are accumulated in the ***data*** array.

xprop uses Fourier space to calculate a step in the interaction picture, using linear transformations that are pre-calculated. There are both linear transformations and momentum dependent terms available. These are pre-calculated by the **xlattice** function, and stored in the *prop* arrays.

User functions

initial is used to initialize each integration in time. This is a user-defined function, which can involve random numbers if there is an initial probability distribution. This creates a stochastic field on the lattice, called ***a.*** The default is **xinitialise**, which sets fields to zero. For sequential simulations, **initials** is used.

step is the algorithm or method computes each space-time point in the lattice. This also generates the random numbers fields at each time-step. It can be user-modified by setting the handle in.step. The default is **xstep**, which uses a central-difference technique.

observe is the observation function whose output is averaged over the ensembles, called from **xpath**. The default, **xobserve**, returns the real amplitudes.

linear is the linear response, including transverse derivatives in space. The default, **xlinear**, sets this to zero.

da is called by **step** to calculate derivatives at every step in the process, including the stochastic terms.

Details of the different parts of the program are given below. Note that the functions **tic**() and **toc()** are called to time each simulation.

6.3 Graphics function

At the end of the loop, global averages and error-bars are calculated. The main functions involved are:

xgraph is called by xSPDE when the ensemble loops finished. The results are graphed and output if required.

xgrpreferences is called by **xgraph** to set the graphics defaults that are not already entered.

Comparison results are calculated if available from the user-specified **in.compare**, an error summary is printed, and the results plotted using the **xgraph** routine, which is a function that graphs the observables. It is prewritten to cover a range of useful graphs, but can be modified to suit the user. The code is intended to cascade down from higher to lower dimension, generating different types of user-defined graphs. Each type of graph is generated once for each specified graphics function.

Results depend on the dimension:

dimension=4 For the highest space dimension, only a slice through $z = 0$ is available. This is then graphed as if it was in three dimensions.

dimension=3 For two dimensions, distinct graphic images of observable *vs x,y* are plotted at *images* time slices. Otherwise, only a slice through $y = 0$ is available. This is then treated as if it was in two dimensions.

dimension=2 For two dimensions, one three-dimensional image of observable *vs x,t* is plotted. Otherwise, only a slice through $x = 0$ is available. This is otherwise treated as in one dimension.

dimension=1 For one dimensions, one image of observable *vs t* is plotted, with data at each lattice point in time. Exact results, error bars and sampling error bounds are included if available.

In addition to time-dependent graphs, the **xgraph** function can generate *images* (3D) and *transverse* (2D) plots at specified points in time, up to a maximum given by the number of time points specified. The number of these can be individually specified for each graphics output. The images available are specified in *imagetype*: 3D perspective plots, grey-scale colour plots and contour plots.

6.4 Error control

The final 2D output graphs will have error-bars if $in.errorchecks = 2$ was specified, which is also the default parameter setting. This is to make sure the final results are accurate. Error-bars below a minimum relative size compared to the vertical range of the plot, specified by the graphics variable *minbar*, are not plotted. There is a clear strategy if the errors are too large.

Either increase the *points*, which gives more plotted points and lower errors, or increase the *steps,* which reduces the step size without changing the graphical resolution. The default algorithm and extrapolation order can be changed, read the xSPDE manual when doing this. Error bars on the graphs can be removed by setting $errorchecks = 1$ or increasing *minbar* in final graphs.

If $ensembles(2) > 1$ or $ensembles(3) > 1$, which allows xSPDE to calculate sampling errors, it will plot upper and lower limits of one standard deviation. If the sampling errors are too large, try increasing $ensembles(1)$, which increases the trajectories in a single thread. An alternative is to increase $ensembles(2)$. This is slower, but is only limited by the compute time, or else to increase $ensembles(3)$, which gives higher level parallelization. Each is limited in different ways; the first by memory, and the second by time, the third by the number of available cores. Sampling error control helps ensures accuracy.

Note that error bars and sampling errors are only graphed for 2D graphs of results vs time. The error-bars are not plotted when they are below a user-specified size, to improve graphics quality. Higher dimensional graphs do not include this, for visibility reasons, but they are still recorded in the data files. Errors caused by the spatial lattice are not checked automatically in the xSPDE code. They must be checked by manually, by comparing results with different transverse lattice ranges and step-size.

7 Arrays

The two important internal types of data that are user accessible are: a, r.

The fields a are complex arrays defined on spatial or momentum grids. Internally, the fields a are just matrices stored on a flattened space lattice, except for temporary transformations to the Fourier domain for calculating interaction picture propagation [8]. Two different types of Fourier representations are used, depending on whether the transformations are for propagation, which requires the fastest possible methods, or whether the transformation is for graphing, which uses a conventional index ordering.

If required, raw ensemble data consisting of all the trajectories a developing in time can be stored and output. This is memory intensive, and is only done if the $in.raw = 1$ option is set.

The lattice data is a structure called, simply, r. It is available to all user-definable routines. The label r is chosen because the lattice parameters have the important function of storing the grid coordinates in space and time. These structures reside in a static internal cell array of inputs and lattice parameters, including the interaction picture transformations, called ***latt***. The data in ***latt*** is different for each simulation in a sequence.

Averaged results for each sequence are stored in either space or Fourier domains, in the array $data$, as determined by the $in.transforms$ vector for the observable. The $data$ arrays obtained in the program as calculations progress are stored in cell arrays, $cdata$, indexed by a sequence index.

The internal spatial grid definitions are as follows:

7.1 Grids in x and k

The algorithms all use a sequence of interaction pictures. Each successive interaction picture is referenced to $t = t_n$, for the n-th step starting at $t = t_n$, so $\mathbf{a}_I(t_n) = \mathbf{a}(t_n) \equiv \mathbf{a}_n$. To understand the interaction picture, we first must understand the xSPDE lattice.

7.1.1 Space lattice

We define the lattice cell size dx_j in the $j - th$ dimension in terms of maximum range R_j and the number of points N_j :

$$dx_j = \frac{R_j}{N_j - 1}\,.$$

Each lattice starts at a value defined by the vector $in.origin$. Using the default values,

the time lattice starts at $t = 0$ and ends at $t = T = r_1$, for $n = 1, \ldots N_j$:

$$t\,(n) = (n-1)dt\,.$$

The j-th coordinate lattice starts at $-r_j/2$ and ends at $r_j/2$, so that, for $n = 1, \ldots N_j$:

$$x_j\,(n) = -R_j/2 + (n-1)dx_j\,.$$

7.1.2 Momentum lattice

The momentum space graphs use a Fourier transform definition so that, for d dimensions:

$$\tilde{\mathbf{a}}\,(\mathbf{k},\omega) = \frac{1}{(2\pi)^{d/2}}\int d\mathbf{x}e^{i(\omega t-\mathbf{k}\cdot\mathbf{x})}\mathbf{a}\,(\mathbf{x},t)$$

In order to match this to the standard definition of a discrete FFT, the j-th momentum lattice cell size dk_j in the $j-th$ dimension is defined in terms of the number of points N_j :

$$dk_j = \frac{2\pi}{dx_j N_j}\,.$$

The momentum range is therefore

$$K_j = (N_j - 1)\,dk_j,$$

while the momentum lattice starts at $-k_j/2$ and ends at $k_j/2$, so that:

$$k_j\,(n) = -K_j/2 + (n-1)dk_j\,.$$

7.2 Computational Fourier transforms

A conventional fast Fourier transform (FFT) is used for the interaction picture (IP) transformations used in computations, as this is fast and simple. In one dimension, this is given by a sum over indices starting with zero, rather than the Matlab convention of one. Hence, if $\tilde{m} = m - 1$:

$$\tilde{a}_{\tilde{n}} = \mathcal{F}\,(a) = \sum_{\tilde{m}=0}^{N-1} a_{\tilde{m}} \exp\left[-2\pi i\tilde{m}\tilde{n}/N\right] \tag{7.1}$$

Suppose the lattice spacing is dx, and the number of lattice points is N, then the maximum range from the first to last point is:

$$R = (N-1)dx$$

We note that the momentum lattice spacing is

$$dk = \frac{2\pi}{Ndx} \tag{7.2}$$

The IP Fourier transform can be written in terms of an FFT as

$$\tilde{\mathbf{a}}\left(\mathbf{k_n}\right)=\prod_j\left[\sum_{\tilde{m}_j}\exp\left[-i\left(dk_jdx_j\right)\tilde{m}_j\tilde{n}_j\right]\right] \tag{7.3}$$

The inverse FFT Fourier transforms automatically divide by the correct factors of $\prod_j N_j$ to ensure invertibility. Note also that due to the periodicity of the exponential function, negative momenta are obtained if we consider an ordered lattice such that:

$$\begin{aligned} k_j &= (j-1)dk \ \ (j\leq N/2) \\ k_j &= (j-1-N)dk \ (j>N/2) \end{aligned}$$

For calculating derivatives and propagating in the interaction picture, the notation D indicates a derivative. To explain, one integrates by parts:

$$D^p\tilde{\mathbf{a}}\left(\mathbf{k}\right)=\left[ik_x\right]^p\tilde{\mathbf{a}}\left(\mathbf{k}\right)=\frac{1}{\left(2\pi\right)^{d/2}}\int d\mathbf{x}e^{-i\mathbf{k}\cdot\mathbf{x}}\left[\frac{\partial}{\partial x}\right]^p\mathbf{a}\left(\mathbf{x}\right) \tag{7.4}$$

This means, for example, that to calculate a one dimensional space derivative in the Linear routine, one uses:

- ∇_x →D.x

Here $D.x$ returns an array of momenta in cyclic order in dimension d as defined above, suitable for an FFT calculation. The imaginary 'i' is not needed to give the correct sign, from Eq (7.4). Instead, it is include in the D array. In two dimensions, the code to return a full two-dimensional Laplacian is:

- $\boldsymbol{\nabla}^2=\nabla_x^2+\nabla_y^2$ →D.x.^2+D.y.^2

Note that the dot in the notation of '.^' is needed to take the square of each element in the array.

7.3 Graphics transforms

The index ordering and normalization used in the standard discrete FFT approach is efficient for interaction picture propagation, but not useful for graphing, since graphics routines prefer the momenta to be monotonic, i.e. in the order:

$$k_j\left(n\right)=-K_j/2+(n-1)dk_j. \tag{7.5}$$

All transforms defined in the observables are obtained from a vector called $in.transforms$, which determines if a given coordinate axis is transformed prior to a given observable being measured. This can be turned on and off independently for each observable. The space and time transforms are defined in the next sub-sections.

7.3.1 Time transforms

We define a Fourier transform in time as:

$$\tilde{\mathbf{a}}(\omega) = \int_0^T \frac{dt}{(2\pi)^{1/2}} \boldsymbol{a}(t) \exp[i\omega t] \tag{7.6}$$

To achieve this in one dimension, note that:

$$dtd\omega = \frac{2\pi}{N}.$$

Defining $\kappa = \omega^{max}/2$:

$$\begin{aligned}
\tilde{\mathbf{a}}(\omega_{\tilde{n}}) &= \frac{dt}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[i(\tilde{n}d\omega - \kappa)(\tilde{m}dt)] a_{\tilde{m}} \\
&= \frac{dt}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[i(\tilde{n}\tilde{m}dtd\omega - \kappa\tilde{m}dt)] a_{\tilde{m}} \\
&= \frac{dt}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[2\pi i\tilde{m}\tilde{n}/N] e^{-i\kappa\tilde{m}dt} a_{\tilde{m}} \\
&= \frac{\sqrt{2\pi}}{d\omega} \times \frac{1}{N} \sum_{\tilde{m}=0}^{N-1} \exp[2\pi i\tilde{m}\tilde{n}/N] e^{-i\kappa t} a_{\tilde{m}}
\end{aligned} \tag{7.7}$$

Hence to get an ordered Fourier transform for graphing data, with the usual physics and mathematics definitions, we must **premultiply** by $e^{-i\kappa t}Ndt/\sqrt{2\pi}$, then take a discrete IFFT. This is taken care of internally in the xSPDE transform routines.

7.3.2 Space transforms

We define a Fourier transform in space as:

$$\tilde{\mathbf{a}}(\mathbf{k}) = \int \frac{dV}{(2\pi)^{d/2}} \boldsymbol{a}(\mathbf{x}) \exp[-i\mathbf{k}\cdot\mathbf{x}] \tag{7.8}$$

To achieve this with an FFT in one dimension, let $\kappa = K/2$, and $\rho = R/2$, and define:

$$\begin{aligned}
\tilde{\mathbf{a}}(k_{\tilde{n}}) &= \frac{dx}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[-i(\tilde{n}dk - K/2)(\tilde{m}dx - R/2)] a_{\tilde{m}} \\
&= \frac{dx}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[-i(\tilde{n}\tilde{m}dxdk - K\tilde{m}dx/2 - R\tilde{n}dk/2 + KR/4)] a_{\tilde{m}} \\
&= e^{-iR(K/2-\tilde{n}dk)/2} \frac{dx}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[-2\pi i\tilde{m}\tilde{n}/N] e^{iK\tilde{m}dx/2} a_{\tilde{m}} \\
&= e^{-iRk/2} \frac{dx}{\sqrt{2\pi}} \sum_{\tilde{m}=0}^{N-1} \exp[-2\pi i\tilde{m}\tilde{n}/N] e^{iKx/2+iKR/4} a_{\tilde{m}}
\end{aligned} \tag{7.9}$$

Hence to get an ordered Fourier transform for graphing data, with the usual mathematical definitions, we must **premultiply** by a phase factor, take a discrete FFT, then **post-multiply** by *another* phase factor. This is taken care of internally in the xSPDE transform routines.

7.4 Fields

In the xSPDE code, the complex vector field a is stored as a complex matrix with dimensions $[fields, lattice]$. Here $lattice$ is the total number of lattice points including an ensemble dimension, to increase computational efficiency:

$$lattice = ensembles(1) \times n.space\,.$$

The total number of space points $n.space$ is given by:

$$n.space = points\,(2) \times \ldots \times points(dimension)\,.$$

The use of a matrix for the fields is convenient in that fast matrix operations are possible in a high-level language.

The $ensembles(1)$ trajectories are used for array-based parallel ensemble averaging. These trajectories are stored in parallel in one array, to allow fast on-chip parallel processing. Distinct stochastic trajectories are also organized at a higher level into a set of $ensembles(2)$ ensembles for statistical purposes, which allows a more precise estimate of sampling error bars. These can also be integrated in parallel using *ensembles(3)* parallel threads.

This hierarchical organization allows allows flexibility in allocating memory and optimizing parallel processing. It is usually faster to have larger values of $ensembles(1)$, but more memory intensive. Using larger values of $ensembles(2)$ is slower, but requires less memory.

In different subroutines it maybe necessary to expand out this array to more easily reference the array structure. The expanded structure is as follows

Array a has dimension: $(fields, ensembles(1), points(2), \ldots points(dimension))$.

Note: Here, $fields$ is the number of field components and $ensembles(1)$ is the number of statistical samples processed as a parallel vector. This can be set to one to save data space, or increased to improve parallel efficiency. Provided no frequency information is needed, the time dimension $points(1)$ is compressed to one during calculations. During spectral calculations, the full length of the time lattice, $points(1)$, is stored, which increases memory requirements.

latt: This includes a propagation array *propagator*, used in the interaction picture calculations. There are two momentum space propagators, for coarse and fine steps respectively, which are computed when they are needed.

7.5 Data

Observables: *data* During the calculation, observables are calculated and averaged over the *ensembles*(1) parallel trajectories in the **xpath** function. The results are added to the earlier results in the array **data**, to create graphs for each observable. At this stage, both the first and second moment is stored, in order to allow calculation of the sampling error in each quantity.

There are *graphs* real observables, which are determined by the number of functions defined in the *observe* cell array. The number of *graphs* may be smaller or larger than the number of vector fields. The observable field includes all the necessary averages over the ensembles.

When step-size checking is turned on using the $errorchecks = 2$ flag, a low resolution field is stored for comparison with a high-resolution field of half the step-size, to obtain the time-step error.

The observable *data* which is stored therefore involves three arrays which are all included in the data array. These are the high resolution means, together with error-bars due to time-steps, and estimates of high-resolution standard deviations due to sampling statistics.

The observable *data* which is plotted therefore includes step-size error bars and plotted lines for the two estimated upper and lower standard deviations, obtained from the statistical moments.

In summary, data from each simulation is stored internally in an array of size

$$errors \times points\,(1) \times n.space \times graphs\,.$$

This is a flattened version of the full data dimension, which is logically

$$errors \times points\,(1) \times \ldots \times points(dimension) \times graphs\,.$$

This is necessary in order to generate outputs at each of the $points(1)$ time slices. Here $errors = 1, 2, 3$ is used to index over the

1. mean value,
2. time-step error-bars and
3. sampling errors

respectively for each space-time point and each graphed function.

Data from each simulation in a sequence is packed into successive cells of an overall cell array *cdata*. This is used to store the total data in a sequence of simulations.

All these fields are resident in memory. They can be re-accessed and replotted, using the **xgraph** function, if required. In summary:

Cell Array *cdata* has dimension: $cdata\{sequence\}$.

Array *data* has dimension: $(errors, points(1), \ldots points(dimension), graphs)$.

The cell index enumerates the sequence number. The first array index $(= 1, 2, 3)$ give the error-checking status of the data. If there is no error-bar checking, the second data array is zero. If there is no sampling error checking, the third data array is zero.

7.6 Raw data

Although the quantity of data generated can be overwhelming, xSPDE can store every trajectory generated if asked to do so.

This raw data is stored in a cell array raw. The array is written to disk using the Matlab file-name, on completion, provided a file name is input.

The cell indices are: the ensemble index, the error-checking index and the sequence index.

Cell Array raw has dimension: $raw\{ensemble, err, seq\}$

Inside each cell is at least one complete space-time $field$ stored as a complex array, with indices for the field index, the sample-space lattice, and the time index. The sample-space lattice structure internal to xSPDE means that a subensemble of individual stochastic fields is integrated in parallel. These are defined as a real or complex array:

Array $field$ has dimension:$(fields, lattice, points(1))$

While this is a lengthy description, and an even larger array, it is also necessary if all the raw data needs to be extracted.

The main utility of the raw data is to provide a platform for further development of analytic tools for third party developers, to treat statistical features not included in the functional tools provided. For example, the basic xSPDE package does not provide histograms of distributions.

8 Algorithms

Stochastic, partial and ordinary differential equations are central to numerical mathematics. Certainly, ordinary differential equations have been known in some form ever since calculus was invented. There are a truly extraordinary number of algorithms used to solve these equations. One program cannot possibly provide all of them.

xSPDE currently provides four built-in choices of algorithm. All built-in methods are defined in an interaction picture. All can be used with any space dimension, including $dimension = 1$, which gives an ordinary stochastic equation. All can be used either with stochastic or with non-stochastic equations. When applied to stochastic equations, the Euler method requires an Ito form of stochastic equation, while the others should be used with the Stratonovich form of these equations. Each uses the interaction picture to take care of exactly soluble linear terms.

If you have a favorite integration method that isn't here, don't panic. User-defined algorithms can be added freely. You can easily add your own. The existing methods are listed below, and the corresponding m-files can be used as a model. Call the routine, for example $'myalgorithm.m'$, set $'in.step = @myalgorithm'$, then adjust the value of $'in.ipsteps'$ if the interaction-picture transform length must be changed to a new value.

Similarly, the interaction-picture transformation, *in.prop*, can also be changed if the built-in choice is not adequate for your needs.

8.1 xSPDE algorithms

The four built-in algorithms provided are:

in.step=@xEuler: the first-order Euler method, a simple first-order explicit approach.

in.step=@xRK2: a second order Runge-Kutta method.

in.step=@xMP: the midpoint method: a semi-implicit, second order algorithm.

in.step=@xRK4: a fourth order Runge-Kutta method, which is a popular ODE solver .

The reader is referred to the literature[5, 6, 7, 9] for more details.

However, a word of caution is in order. For stochastic equations, which are non-differentiable, the classifications of convergence order should be taken *cum grano salis.* In other words, don't believe it. Stochastic convergence is a complex issue, and the usual rules of calculus don't apply. This is because stochastic noise is non-differentiable. It has relative fluctuations proportional to $1/\sqrt{dtdV}$, for noise defined on a lattice with temporal cell-size dt and spatial cell-size dV. Hence the usual differentiability and smoothness

properties required to give high-order convergence for standard Runge-Kutta methods are simply not present.

All is not completely lost however, since xSPDE will attempt to estimate both the step-size and the sampling error, so you can check convergence yourself.

8.2 Euler

This is an explicit Ito-Euler method using an interaction picture. While very traditional, it is not generally recommended except for testing purposes. If it is used, very small step-sizes will generally be necessary to reduce errors to a usable level.

This is because it is is only convergent to first order, and therefore tends to have large errors. It is designed for use with an Ito form of stochastic equation. It requires one IP transform per step ($ipsteps = 1$). To get the next time point, one calculates:

$$\begin{aligned} \Delta\mathbf{a}_n &= \Delta t\mathbf{D}\left[\mathbf{a}_n\right] \\ \mathbf{a}_{n+1} &= \mathbf{P}\left(\Delta t\right)\cdot\left[\mathbf{a}_n + \Delta\mathbf{a}_n\right] \end{aligned} \tag{8.1}$$

8.3 Second order Runge-Kutta

This is a second order Runge-Kutta method using an interaction picture [8]. It is convergent to second order in time for non-stochastic equations, but for stochastic equations it can be more slowly convergent than the midpoint method. It requires two IP transforms per step, but each is a full time-step long ($ipsteps = 1$).

To get the next time point, one calculates:

$$\begin{aligned} \bar{\mathbf{a}} &= \mathbf{P}\left(\Delta t\right)\cdot\left[\mathbf{a}_n\right] \\ \mathbf{d}^{(1)} &= \Delta t\mathbf{P}\left(\Delta t\right)\cdot\mathbf{D}\left[\mathbf{a}_n\right] \\ \mathbf{d}^{(2)} &= \Delta t\mathbf{D}\left[\bar{\mathbf{a}} + \mathbf{d}^{(1)}\right] \\ \mathbf{a}_{n+1} &= \bar{\mathbf{a}} + \left(\mathbf{d}^{(1)} + \mathbf{d}^{(2)}\right)/2 \end{aligned} \tag{8.2}$$

8.4 Midpoint

This is an implicit midpoint method using an interaction picture. It gives good results for stochastic [5] and stochastic partial differential equations [7]. While it is only convergent to second order in time for non-stochastic equations, it is strongly convergent and robust. It requires two half-length IP transforms per step ($ipsteps = 2$).

To get the next time point, one calculates a midpoint derivative iteratively at $\bar{\mathbf{a}}^{(i)}$, usually with three iterations:

$$\begin{aligned}
\bar{\mathbf{a}}^{(0)} &= \mathbf{P}\left(\frac{\Delta t}{2}\right)\cdot[\mathbf{a}_n] \\
\bar{\mathbf{a}}^{(i)} &= \bar{\mathbf{a}}^{(0)} + \frac{\Delta t}{2}\mathbf{D}\left[\bar{\mathbf{a}}^{(i-1)}\right] \\
\mathbf{a}_{n+1} &= \mathbf{P}\left(\frac{\Delta t}{2}\right)\cdot\left[2\bar{\mathbf{a}}^{(i)} - \bar{\mathbf{a}}^{(0)}\right]
\end{aligned} \tag{8.3}$$

8.5 Fourth order Runge-Kutta

This is a fourth order Runge-Kutta method using an interaction picture [8]. It is convergent to fourth order in time for non-stochastic equations, but for stochastic equations it can be more slowly convergent than the midpoint method. It requires four half-length IP transforms per step (*ipsteps* = 2). To get the next time point, one calculates four derivatives sequentially:

$$\begin{aligned}
\bar{\mathbf{a}} &= \mathbf{P}\left(\frac{\Delta t}{2}\right)\cdot[\mathbf{a}_n] \\
\mathbf{d}^{(1)} &= \frac{\Delta t}{2}\mathbf{P}\left(\frac{\Delta t}{2}\right)\cdot\mathbf{D}\left[\mathbf{a}_n\right] \\
\mathbf{d}^{(2)} &= \frac{\Delta t}{2}\mathbf{D}\left[\bar{\mathbf{a}} + \mathbf{d}^{(1)}\right] \\
\mathbf{d}^{(3)} &= \frac{\Delta t}{2}\mathbf{D}\left[\bar{\mathbf{a}} + \mathbf{d}^{(2)}\right] \\
\mathbf{d}^{(4)} &= \frac{\Delta t}{2}\mathbf{D}\left[\mathbf{P}\left(\frac{\Delta t}{2}\right)\left[\bar{\mathbf{a}} + 2\mathbf{d}^{(3)}\right]\right] \\
\mathbf{a}_{n+1} &= \mathbf{P}\left(\frac{\Delta t}{2}\right)\cdot\left[\bar{\mathbf{a}} + \left(\mathbf{d}^{(1)} + 2\left(\mathbf{d}^{(2)} + \mathbf{d}^{(3)}\right)\right)/3\right] + \mathbf{d}^{(4)}/3
\end{aligned} \tag{8.4}$$

This might seem like the obvious choice, having the highest order. However, it can actually converge at a range of apparent rates, depending on the relative importance of stochastic and non-stochastic terms. Due to its reliance on differentiability, it may converge more slowly than the midpoint method with stochastic terms present.

The actual error is best judged by measuring it, as explained next.

8.6 Convergence checks

To check convergence, xSPDE repeats the calculations at least twice for checking step-sizes, and many times more in stochastic cases. *If you think this is too boring and slow, turn it off.* However, you won't know your errors!

Whatever the application, you will find the error-estimates useful. If the errors are too large, and this is relative to the application, you should decrease the time-steps or increase the number of samples. Which to do entirely depends on the type of error. In

xSPDE, the step-size error due to finite time-step sizes is called the 'step' error. The sampling error due to finite samples of trajectories is called the 'sample' error. The maximum value of each of these, calculated over the set of all computed observables, is printed out at the end of the run.

Where there is 2D graphical output, the error bars give the step-size error, if you have $check = 2$. To distinguish the error types, two lines are graphed for an upper and lower standard deviation departure from the mean, indicating the sampling error. This is only plotted if *ensemble* is greater than one, preferably at least $10 - 20$ to give reliable estimates.

Note that the sample error is usually reasonably accurate. It occasionally may underestimate errors for pathological distributions. The step error is generally the more cautious of the two, and tends to overestimate errors. Neither should be relied as more than a rough guide.

As a check, the code allows users to graph a defined 2D exact result, if known, for comparison and testing purposes. These are graphed using dashed lines. This facility can be turned on or off for each observable using Boolean variables. This can be useful even if no exact result is known, but there is a known conservation law.

In summary, there are three types of convergence checks, all of which appear in the output as printed maximum values and projected two-dimensional graphs:

- Error bars indicate the error due to finite step-size
- Upper and lower solid lines indicate the $\pm\sigma$ sampling error bounds
- Dashed lines indicate comparison values, which are useful when there are exact results for testing

8.7 Extrapolation order and error bars

For checking step-size errors, xSPDE allows the user to specify $errorchecks = 2$, which is the default option. This gives one integration at the specified step-size, and one at half the specified step-size. The data is plotted at the fine step-size. The standard error-bar, with no extrapolation, has a half-size equal to the difference of fine and coarse step graphed results.

To allow for extrapolation, xSPDE allows user input of an assumed extrapolation order called *order*. If this is done, and errorchecks are set to 2 to allow successive integration with two different step-sizes, the output of all data graphed will be extrapolated to the specified order. In this case, the error bar half-size is set to the difference of the fine estimate and the *extrapolated* estimate.

Extrapolation is a well-known technique for improving the accuracy of a differential equation solver. Suppose an algorithm has a result with a known convergence order n. This means that for small enough step-size, integration results $R(dt)$ with step-size dt have an error of size dt^n, that is:

$$R(dt) = R_0 + E(dt) = R_0 + k.dt^n . \tag{8.5}$$

Hence, from two results at different values of dt, differing by a factor of 2, one would obtain

$$\begin{aligned} R_1 &= R(dt) = R_0 + k.dt^n \\ R_2 &= R(2dt) = R_0 + 2^n k.dt^n . \end{aligned} \tag{8.6}$$

The true result, extrapolated to the small-step size limit, is therefore given by giving more weight to the fine step-size result, while *subtracting* from this a correction due to the coarse step-size calculation:

$$R_0 = \frac{[R_1 - R_2 2^{-n}]}{[1 - 2^{-n}]} . \tag{8.7}$$

Thus, for example, if we define a factor ϵ as

$$\epsilon(n) = \frac{1}{[2^n - 1]} = \left(1, \frac{1}{3}, \frac{1}{7} \ldots\right) , \tag{8.8}$$

then the true results are obtained from extrapolation to zero step-size as:

$$R_0 = (1 + \epsilon) R_1 - \epsilon R_2 . \tag{8.9}$$

The built-in algorithms have convergence order as ordinary differential equation integrators of $1, 2, 2, 4$ respectively, and should converge to this order at small step-sizes.

However, the situation is not as straightforward for stochastic equations. First order convergence is always obtainable stochastically. In addition, second order convergence is generally obtainable with the midpoint algorithm, although this is not guaranteed: it depends on the precise noise term. However, the Runge-Kutta algorithms used do **not** converge to the standard ODE order for stochastic equations. Hence extrapolation should be used with extreme caution in stochastic calculations.

While extrapolated results are usually inside those given by the default error-bars, **extrapolation with too high an order can under-estimate the resulting error bars.** Therefore, xSPDE assumes a cautious default order of $order = 1$. Note that one can set $order = 0$ to obtain fine resolution values and error bars without extrapolation, but this is generally less accurate.

8.8 Sampling errors

Sampling error estimation in xSPDE uses sub-ensemble averaging. Ensembles are specified in three levels. The first, *ensemble(1)*, is called the number of samples for brevity. All computed quantities returned by the **observe** functions are first averaged over the samples, which are calculated efficiently using a parallel vector of trajectories. By the central limit theorem, these sample averages are distributed as a normal distribution at large sample number.

Next, the sample averages are averaged **again** over the two higher level ensembles, if specified. This time, the variance is accumulated. The variance of these distributions is

used to estimate a standard deviation in the mean, since each computed quantity is now a normally distributed result. This method is applied to all the *graphs* observables. The two lines generated represent $\bar{o} \pm \sigma$, where o is the observe function output, and σ is the standard deviation in the mean.

The highest level ensemble, *ensemble(3),* is used for parallel simulations. This requires the Matlab parallel toolbox. Either type of high-level ensemble, or both together, can be used to calculate sampling errors.

Note that one standard deviation is not a strong bound; errors are expected to exceed this value in 32% of observed measurements. Another point to remember is that stochastic errors are often correlated, so that a group of points may all have similar errors due to statistical sampling.

9 Extensibility

This chapter will treats the extensibility of xSPDE.

9.1 Open object-oriented architecture

As well as extensibility through sequences, which was described in Chapter 3, in section 3.5, the open architecture of xSPDE allows functional extensions.

This further type of extensibility permits user definable functions to be specified in the *in* metadata structures. These functions have default values in xSPDE, and can simply be used as is. It is also possible to have user defined functions that satisfy the interface definitions instead. This is achieved simply by including the relevant function handles and parameters in the input metadata.

This input metadata includes both data and methods acting on the data, in the tradition of object-oriented programs. Yet there is no strict class typing. Users are encouraged to adapt the xSPDE program by adding more input parameters and methods to the input structures. Internal parameters and function handles stored in the lattice structure r are available to all user-defined simulation functions. Note that use of pre-existing reserved names is not advisable - see tables below.

Such unorthodox object orientation is deliberate.

The xSPDE software architecture is intended to be easily extended, and users are encouraged to develop their own libraries. Because this may require new functions and parameters, the internal data architecture is as open as possible.

For example, to define your own integration function, include in the xSPDE input the line:

```
in.step=@Mystep;
```

Next, include anywhere on your Matlab path, the function definition, for example:

```
function a = Mystep(a,xi,dt,r)
% a = Mystep(a,xi,dt,r) propagates a step my way.
..
a = ...;
end
```

9.2 Table of xSPDE metadata

The input cell array, *input*, includes input data in structures *in*. This data is also passed to the *xgraph* function. The inputs are numbers, vectors, strings, functions and cell arrays. All xSPDE metadata has preferred values, so only changes from the preferences need to be input. The resulting data is stored internally as a sequence of structures in a cell array, to describe the simulation sequence.

Simulation metadata, including all preferred default values that were used in a particular simulation, is also stored for reference in any xSPDE output files. This is done in both the *.mat* and the *.h*5output files, so the entire simulation can be easily reconstructed or changed.

Note that inputs can be numbers, vectors, strings or cells arrays. To simplify the inputs, some conventions are used, as follows:

- **All input data has default values**
- **Vector inputs of numbers are enclosed in square brackets, [...].**
- **Where multiple inputs of strings, functions or vectors are needed they should be enclosed in curly brackets, {...}, to create a cell array.**
- **Vector or cell array inputs with only one member don't require brackets.**
- **Incomplete or partial vector or cell array inputs are filled in with the last applicable default value.**
- **New function definitions can be just handles pointing elsewhere.**

9.2.1 Table of user functions

The user-definable functions, output field dimensions and calling arguments are:

Label	Output dimension	Arguments	Purpose
**da*	d.a	*(a,z,r)*	Stochastic derivative
**initial*	d.a	*(v,r)*	Function to initialize fields
**linear*	d.a	(D,r)	Linear interaction picture function
rfilter	d.random2	(r)	Random input filter function in k-space
nfilter	d.noise2	(r)	Noise filter function in k-space
transfer	d.a	*(v,a0,r,r0)*	Function to initialize fields in sequences
step	d.a	(a,z,dt,r)	Function to calculate one time-step
grid	d.r	(r)	Grid calculator for lattice
prop	d.a	(a,r)	Interaction picture (IP) propagation
propfactor	d.a	(nc,r)	IP propagator array calculation
observe	d.int	*(a,r)*	Observable function cell array
compare	points(1)	(t,in)	Comparison function cell array

9.2.2 Constants for *in* structure

Label	Type	Typical value	Description
version	string	*'xSPDEv1.01'*	Current version number (added automatically)
name	string	"	Simulation name
dimension	integer	1	Space-time dimension
fields	integer	1	Number of stochastic fields
ranges	vector	10	Range of coordinates in [t,x,y,z]
origin	vector	0	Origin of coordinates in [t,x,y,z]
points	vector	51	Output lattice points in [t,x,y,z]
noises	vector	*[1 0]*	Number of noise fields in [x,k]
randoms	vector	*[1 0]*	Initial random fields in [x,k]
ensembles	vector	*[1 1 1]*	Ensembles used for averaging
steps	integer	*1*	Integration steps per output point
iterations	integer	*4*	Maximum iterations
order	integer	*1*	Extrapolation order
errorchecks	integer	*2*	Number of error-checking cycles
seed	integer	*0*	Seed for random number generator
file	string	*'f.mat' or 'f.h5'*	File-name for HDF5 or Matlab data file
ipsteps	integer	*2*	IP transforms per time-step
graphs	integer	*1*	Number of observables
print	integer	1	Print switch, 1 for medium output
c	any	-	User specified static constants or structures
olabels	string cells	*{'a_1',..}*	Observable labels
transforms	vector cells	*{[0 0 0 0],..}*	Fourier transforms in [t,x,y,z]
raw	integer	0	Raw trajectory switch, 1 for output

9.2.3 Graphics data for *in* structure

Label	Type	Typical value	Description
minbar	cell	*{0.01,..}*	Minimum relative error-bar size
font	cell	{18,..}	Font size for graph labels
xlabels	cell	*{'t' 'x' 'y' 'z'}*	Strings for x-axis labels
klabels	cell	*{'\omega' 'k_x' 'k_y' 'k_z'}*	Transformed axis labels
headers	cell	{0,..}	Switch for graph headers
images	cell	{0,..}	Number of movie style images
imagetype	cell	{0,..}	Type of movie style images
transverse	cell	{0,..}	Number of transverse plots
pdimension	cell	{2,..}	Maximum plot dimensions
**compare*	function cell	{@(t,in) t^2}	Comparison functions

9.2.4 Internal xSPDE parameters in 'r' structure

Internally, xSPDE data is stored in a cell array, *latt*, of structures r, which is passed to functions. This includes all the data given above inside the *in* structure. In adition, it includes the table of computed parameters given below.

User application constants and parameters should not be reserved names; *'in.c'* and all names starting with '*in.c*' will always be available in all versions of xSPDE.

Label	Type	Typical value	Description
t			Current value of time, t
x, y, z	array	-	Grid of x, y, z
kx, ky, kz	array	-	Grid of $[k_x, k_y, k_z$
dx	vector	*0.2*	Steps in $[t, x, y, z]$
dk	vector	*0.6160*	Steps in $[\omega, k_x, k_y, k_z]$
dt	double	*0.2000*	Output time-step
V	real	*1*	Spatial lattice volume
K	real	*1*	Momentum lattice volume
dV	real	*1*	Spatial cell volume
dK	real	*1*	Momentum cell volume
xc	vector cells	*{[0,... 10]}*	Space-time coordinate axes
kc	vector cells	*{[0,..15,-15...]}*	Computational axes in$[\omega, k_x, k_y, k_z]$
gk	vector cells	*{[-15,... 15]}*	Graphics axes in$[\omega, k_x, k_y, k_z]$
kr	vector	*30*	Range in$[\omega, k_x, k_y, k_z]$
wtph	vector	-	Frequency phase-factors
s.dx	double	1	Initial stochastic normalization
s.dxt	double	*3.1623*	Propagating stochastic normalization
s.dk	double	1	Initial k stochastic normalization
s.dkt	double	*3.1623*	Propagating k stochastic normalization
n.space	integer	1	Number of spatial lattice points
n.lattice	integer	1	Total lattice: ensembles(1) x n.space
n.ensemble	integer	1	ensembles(2) x ensembles(3)
n.random	integer	1	Number of initial random fields
n.noise	integer	1	Number of noise fields
d.int	vector	*[1 1]*	Dimensions for lattice integration
d.a	vector	*[1 1]*	Dimensions for a field (flattened)
d.r	vector	*[1 1]*	Dimensions for coordinates (flattened)
d.ft	vector	*[1 1 1]*	Dimensions for field transforms
d.k	vector	*[0 1 1]*	Dimensions for noise transforms
d.obs	vector	*[1 1 1 1]*	Dimensions for observations
d.data	vector	*[1 3 51 1]*	Dimensions for average data (flattened)
d.raw	vector	*[1 1 51 1]*	Dimensions for raw data (flattened)

Bibliography

[1] C. W. Gardiner, *Handbook of Stochastic Methods: for Physics, Chemistry and the Natural Sciences* (Springer 2004).

[2] G. R. Collecutt, P. D. Drummond, *Xmds: eXtensible multi-dimensional simulator.* Comput. Phys. Commun. **142**, 219-223 (2001).

[3] Graham R. Dennis, Joseph J. Hope, Mattias T. Johnsson, *XMDS2: Fast, scalable simulation of coupled stochastic partial differential equations,* Computer Physics Communications **184**, 201–208 (2013).

[4] http://sourceforge.net/projects/xmds/

[5] P. D. Drummond and I. K. Mortimer, *Computer simulations of multiplicative stochastic differential equations.* J. Comp. Phys. **93**, 144-170 (1991).

[6] P.E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations* (Springer, 1995).

[7] M. J. Werner and P. D. Drummond, *Robust algorithms for solving stochastic partial differential equations.* J. Comp. Phys. **132**, 312-326 (1997).

[8] B. M. Caradoc-Davies, *Vortex dynamics in Bose-Einstein condensates* (Doctoral dissertation, PhD thesis, University of Otago (NZ), 2000).

[9] Desmond J. Higham, *An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations*, SIAM Review **43**, 525–546 (2001).