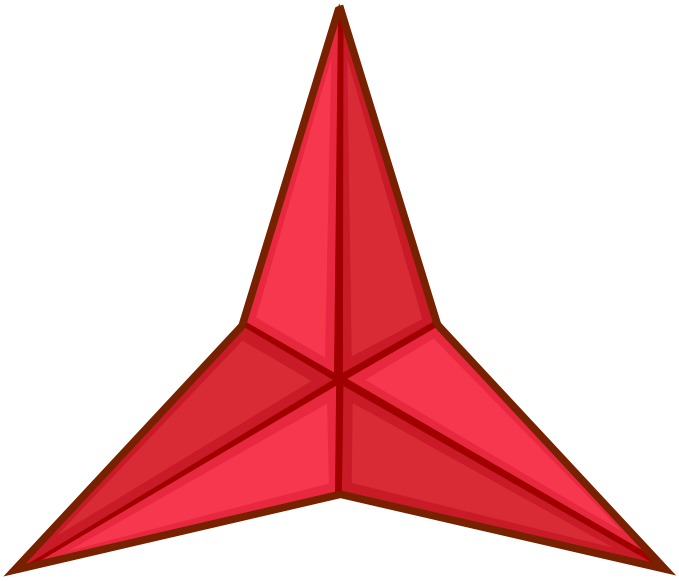

XSPDE Documentation

Release 3.2

Peter Drummond

Dec 12, 2019

CONTENTS

Peter D. Drummond, Simon Kiesewetter

xSPDE was developed at the Centre for Quantum and Optical Science, Swinburne University of Technology, Melbourne, Victoria, Australia.

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.

CHAPTER

ONE

INTRODUCTION TO XSPDE3.2

Stochastic equations are equations with random noise terms [Gardiner2004]. 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.

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 [Gardiner2004]. 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 of xSPDE with an *.m* ending are written in Matlab, an interpreted scientific language of The Mathworks Inc, and are compatible with Octave, an open-source clone of Matlab. This is best regarded as a prototyping platform. A code for new applications can be quickly developed and tested. This is not 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.

Readers of this document may also wish to try XMDS [Colecutt2001], and its successor, XMDS2 [Dennis2013], which are similar programs using XML input files.

Applications

There are many types of stochastic equations, 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 can also display multiple trajectories and calculate probabilities of any function of the fields.

It uses sub-ensemble averaging and extrapolation to obtain accurate error estimates.

Source code

The source code for xSPDE is available at github.com/peterddrummond/xspde_matlab. and at: mathworks.com/matlabcentral/fileexchange/58771-xspde.

Note: xSPDE is distributed without guarantee, under the BSD open-source license. There are no currently known bugs. Please test it yourself before use, all feedback and bug reports are welcome.

CHAPTER

TWO

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 creating fields in the input structure. A complete list of parameters and their uses is given in *Public API*.

The simplest way to use xSPDE is via the interactive Matlab command window, illustrated in this chapter. In this mode of operation, any required parameters are entered into the input structure `in`, and then the command `xspde(in)` runs and graphs the simulation. Note that one must enter `clear` first to erase previous data in the Matlab workspace, unless the previous data is being recycled.

2.1 Stochastic equations

An ordinary stochastic equation [Kloeden1995] for a real or complex vector $\boldsymbol{a}$ is:

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

Here $\boldsymbol{A}$ is a vector function, $\underline{\mathbf{B}}$ a matrix, and $\boldsymbol{\zeta}$ is a real Gaussian distributed noise vector such that $\langle\boldsymbol{\zeta}\rangle = 0$, and :

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

2.2 xSPDE simulations

To simulate a stochastic equation like this interactively with xSPDE, first make sure Matlab path is pointing to the xSPDE folder and type `clear` to clear old data.

Next, enter the xspde parameters (see *Public API*) into the command window, as follows:

```
in.label1 = <parameter1>
in.label2 = ...
in.da = @(a,w,r) <expression for da/dt>
xspde(in)
```

- The notation `in.label = parameter` creates a field in the structure `in` (which is created dynamically, if it has not been defined before).
- The notation `@(..)` is the Matlab shorthand for an anonymous function.
- The parameters passed to the *`da()`* function are: `a`, the stochastic variable; `w`, the random noise; and `r`, the input structure with additional coordinates and parameters.
- *`xspde()`* is called with the input structure as the only argument.

- parameters or functions that are omitted are replaced with default values.
- a sequence of simulations requires an input list: *{in1,in2..}*.

Once the simulation is completed, xSPDE will generate graphs of averages of any required observables or moments. If needed, simulated data is stored in specified files for later use, which requires a file-name to be entered.

2.3 Interactive examples

2.3.1 The random walk

The first example is the simplest possible stochastic equation:

$$\dot{a} = w(t),$$

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

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

- Here *da()* defines the derivative function. The notation @(a,w,r) defines an inline function. In this case the derivative equals the noise `w`. Other parameters have their default values.
- The last argument of xspde user functions is `r`, containing the parameters required for the simulation.

2.3.2 Laser quantum noise

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

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

where $\zeta = (w_1 + iw_2)$, so that:

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

Here the coefficient b describes the quantum noise of the laser, and is inversely proportional to the equilibrium photon number. An xSPDE script in Matlab is given below, for the case of $b = 0.01$, with an output graph in Fig. 2.2. Note the use of the clear command here to clean up the Matlab workspace before the start.

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

Note that:

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

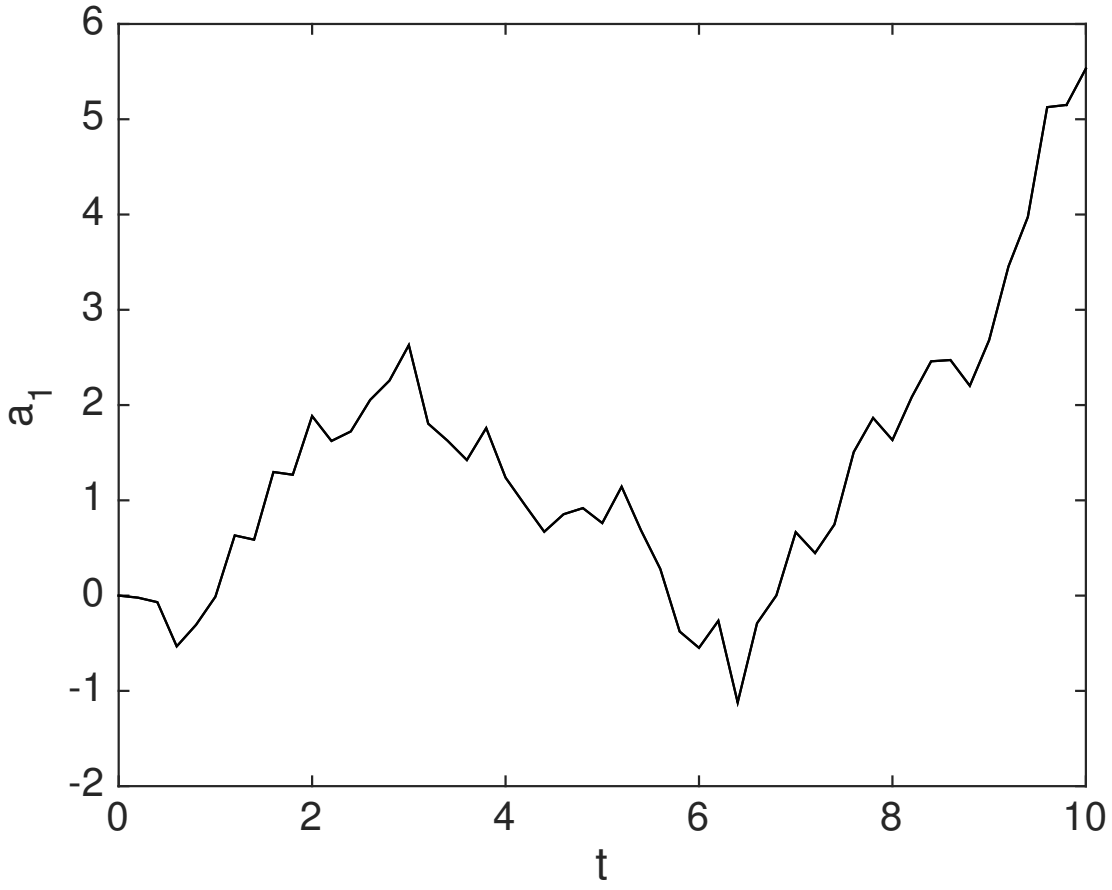

Fig. 2.1: The simplest case: a random walk.

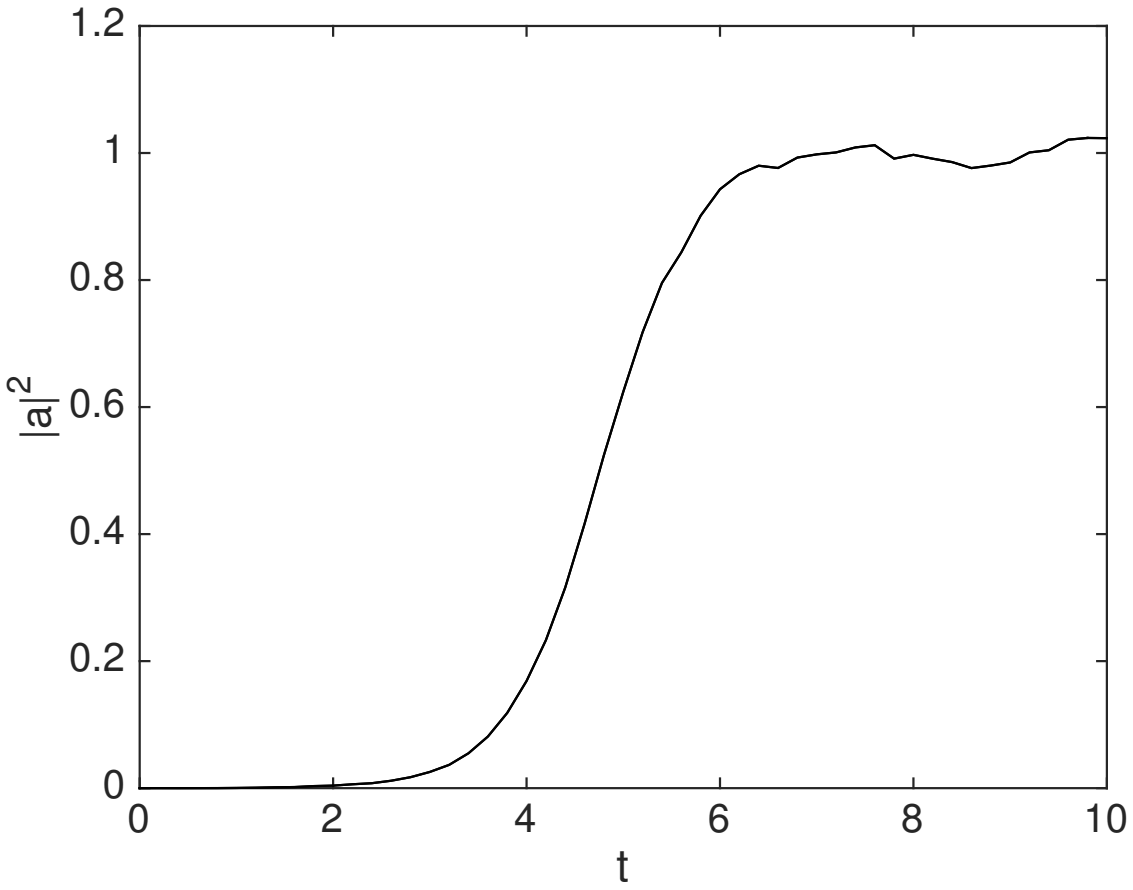

Fig. 2.2: Simulation of the stochastic equation describing a laser turning on.

2.4 Ito and Stratonovich equations

The xSPDE toolbox is primarily designed to treat Stratonovich equations [Gardiner2004], which are the broad-band limit of a finite band-width random noise equation, with derivatives evaluated at the midpoint in time of a time-step.

An equivalent type 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:

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

Here:

$$\left\langle dw_i\left(\boldsymbol{x}\right) dw_j\left(\boldsymbol{x}'\right)\right\rangle = \delta_{ij} dt.$$

When 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

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

Provided the noise coefficient B is constant - which is called additive noise - there is no real difference between the two types of equation. Otherwise, it is essential to know which type of stochastic equation it is, in order to get unambiguous results!

2.4.1 Financial calculus

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

$$da = \mu a\, dt + \sigma a\, dw,$$

where $\left\langle dw^2\right\rangle = dt$. 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:

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

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.

```
clear
in.initial = @(rv,r) 1;
in.da = @(a,w,r) -0.4*a+a*w;
xspde(in)
```

- Here `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 *Averaging and projects*.

This graph is of a single stochastic realisation. Generation of averages is also straightforward. This is described in *Averaging and projects*.

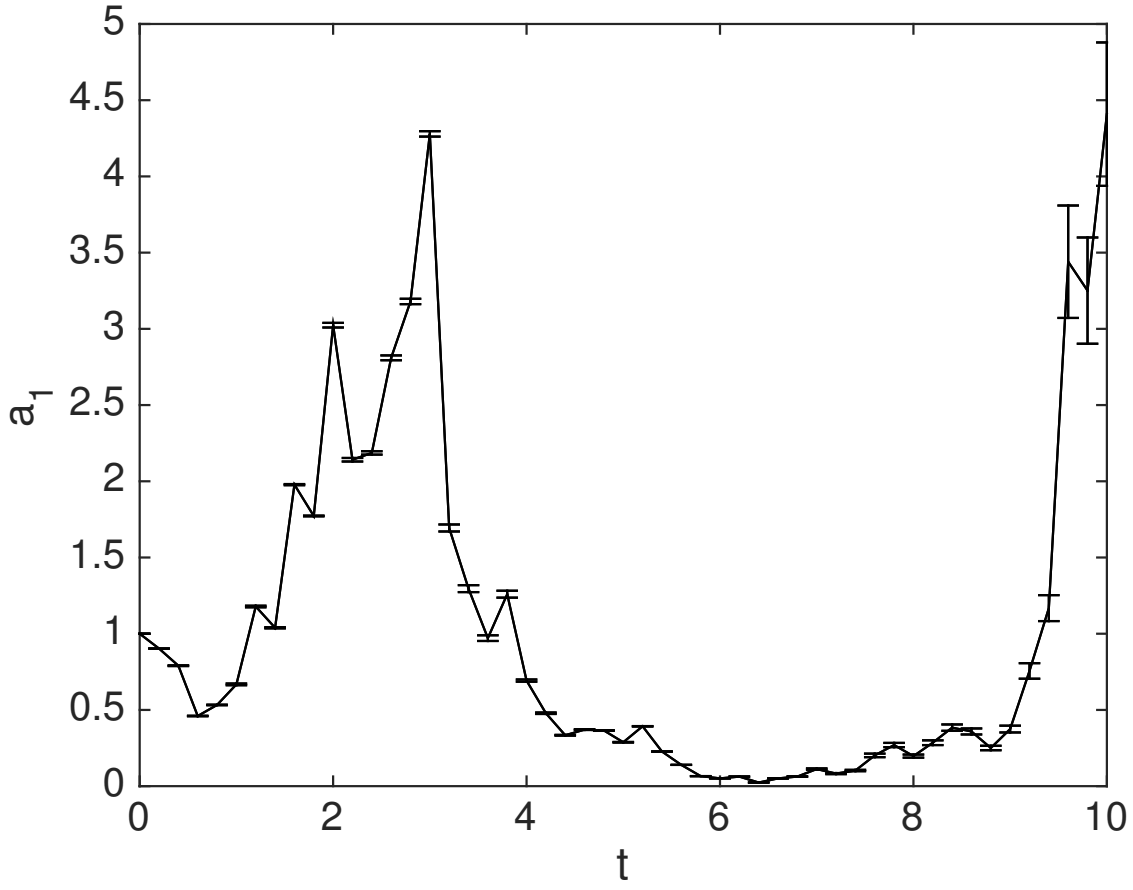

Fig. 2.3: Simulation of the Black-Scholes equation describing stock prices.

2.5 Stochastic partial differential equations

More generally, xSPDE solves [Werner1997] a stochastic partial differential equation for a complex vector field defined with arbitrary transverse dimension. The total dimension is then input as $d = 2, 3, ...$ Equations of this type occur in many disciplines, including biology, chemistry, engineering and physics. They are in differential form as

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

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

$$\left\langle \zeta_i^x\left(t, \boldsymbol{x}\right) \zeta_j^x\left(t, \boldsymbol{x}'\right)\right\rangle = \delta\left(\boldsymbol{x} - \boldsymbol{x}'\right) \delta\left(t - t'\right) \delta_{ij}$$
$$\left\langle \zeta_i^k\left(t, \boldsymbol{k}\right) \zeta_j^k\left(t, \boldsymbol{k}'\right)\right\rangle = f(\boldsymbol{k}) \delta\left(\boldsymbol{k} - \boldsymbol{k}'\right) \delta\left(t - t'\right) \delta_{ij}.$$

Note that the x and k noise term for each value of the index are generated from the same underlying stochastic process. This is necessary because there are some equations that require both a filtered and unfiltered noise generated from the same underlying random number distribution. If these correlations are not wanted, and the noises are required to be independent, then different noise indices must be used.

Transverse boundary conditions are assumed periodic as the default option, which allows the use of efficient spectral Fourier transform propagation codes. Other types of boundary conditions available are Neumann boundaries with zero normal derivatives, and Dirichlet boundaries with zero fields at the boundary. These require the use of finite difference methods. The boundary type can be individually specified in each axis direction. The term $\underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right]$ may be omitted, as space derivatives can also be treated directly in the derivative function, and this is necessary with Neumann or Dirichlet boundaries. The momentum filter $f(\boldsymbol{k})$ is an arbitrary user-specified function, allowing for spatially correlated noise.

To treat stochastic partial differential equations or SPDEs, in the most efficient way, the equations are divided into the first two terms, which are essentially an ordinary stochastic equation, and the last term which is assumed linear, and therefore gives a linear partial differential equation:

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

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 $\tilde{\boldsymbol{a}}\left(\boldsymbol{k}\right) = \mathcal{F}\left[\boldsymbol{a}\left(\mathbf{x}\right)\right]$ is the Fourier transform of $\boldsymbol{a}$, we simply define:

$$\tilde{\boldsymbol{a}}(\boldsymbol{k}, dt) = \mathcal{P}\left(\boldsymbol{k}, dt\right) \tilde{\mathbf{a}}_I\left(\boldsymbol{k}, dt\right)$$

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

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

The total derivative in the interaction picture is the xSPDE derivative function `da()`:

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

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

Note that there are many other types of partial differential equation that can be treated with xSPDE, even if the interaction picture method doesn't apply. This occurs when there are nonlinear functions with arbitrary derivatives, or derivatives that are non-diagonal in the vector indices. For these cases, the space derivatives are evaluated inside the derivative term `da()`:. If there are higher order time derivatives as well, these can be re-expressed as a set of first-order time derivatives, provided the problem is an initial-value problem.

2.5.1 Symmetry breaking

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

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

where

$$\langle \zeta(x)\zeta^*(x')\rangle = 2\delta\left(t - t'\right)\delta\left(x - x'\right).$$

An xSPDE script is given below, for parameter values of $b = 0.001$ and $c = 0.01$, with the output graphed in Fig. 2.4. Note that in the graph, the range `-5<x<5` is the default xSPDE coordinate range, while the `.*` notation is used in functions here, as fields require element-wise multiplication.

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

Here:

- *dimension* is the space-time dimension, with an $x - t$ plot given here.
- *steps* gives the integration steps per plot-point, for improved accuracy.
- *linear* is the linear operator — an imaginary laplacian
- `r.Dx` indicates a derivative operation, $\partial/\partial x$. See the reference entry for *linear* for more information.

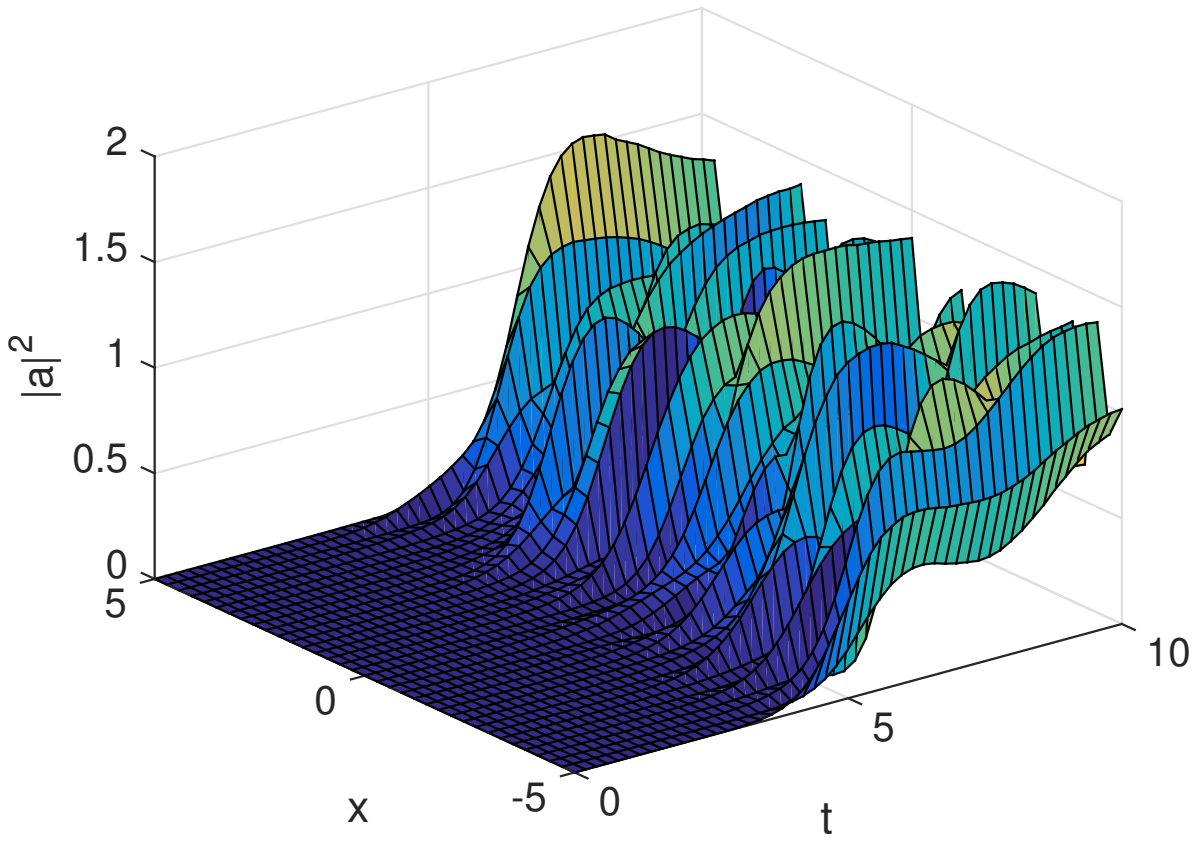

Fig. 2.4: Simulation of the stochastic equation describing symmetry breaking in two dimensions. Spatial fluctuations are caused by the different phase-domains that interfere. The graph obtained here is projected onto the $y = 0$ plane.

CHAPTER

THREE

AVERAGING AND PROJECTS

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,input,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 = 'Intensity';
```

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) * ensembles(2) * 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:

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

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:

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

3.3 xSPDE projects

An XPDE session can either run simulations interactively, described in *Interactive xSPDE*, 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 = project()
    in.label1 = parameter1;
    in.label2 = parameter2;
    ...
    xspde(in)
end
```

For standard graph generation, the script input or project function should end with the combined function *xspde()*. 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.

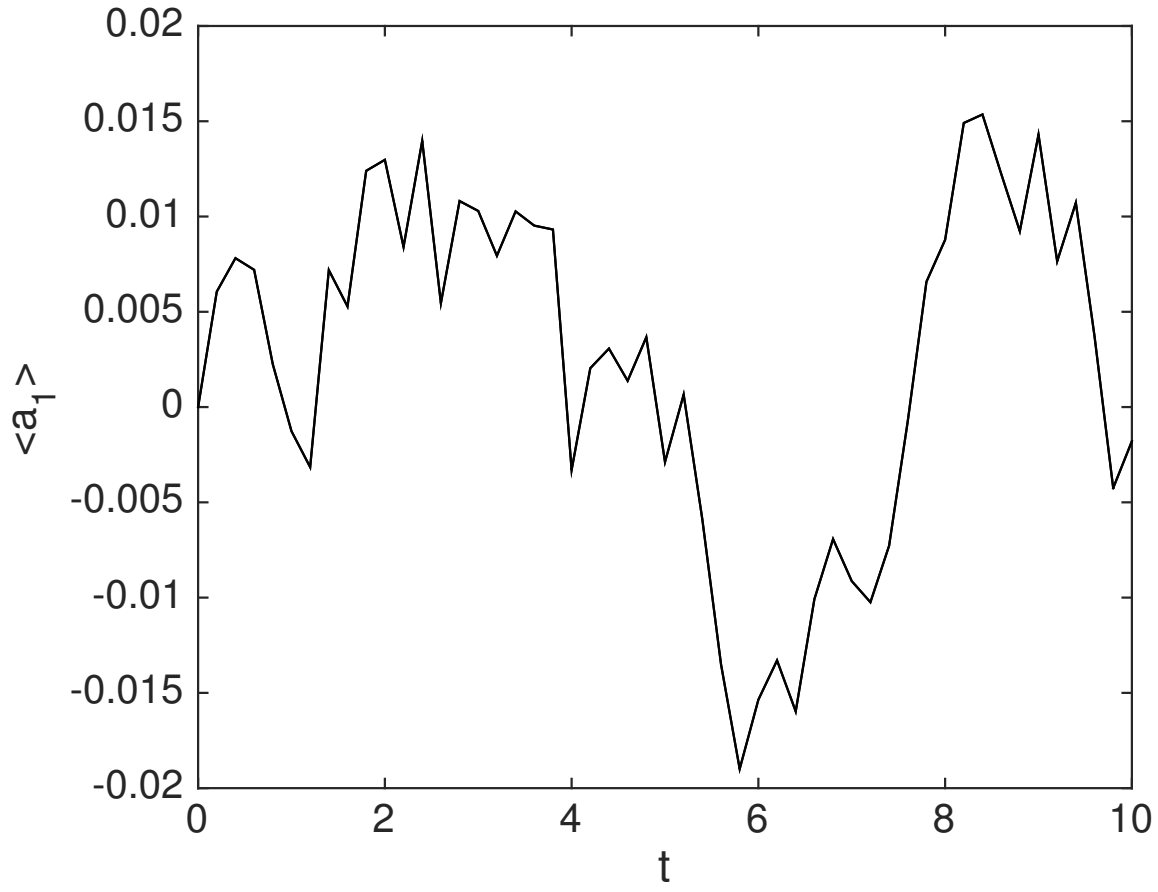

Fig. 3.1: The average random walk with 10^4 trajectories.

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:

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

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 = @(rv,~) 1+0*rv;
    in.da = @(a,w,r) i*a.*w;
    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.

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.
- *`name`* gives a name to identify the project.
- *`ensembles`* specifies 400 samples in a parallel vector, repeated 16 times in series.
- *`initial()`* initializes the input to ones; the noise `rv` is used as it has the same lattice dimension as the `a` field.
- *`da()`* is the function, $da/dt = iaw$, that specifies the equation being integrated.
- *`olabels`* is a cell array with a label for the variable `a` that is averaged.
- *`xspde()`* 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

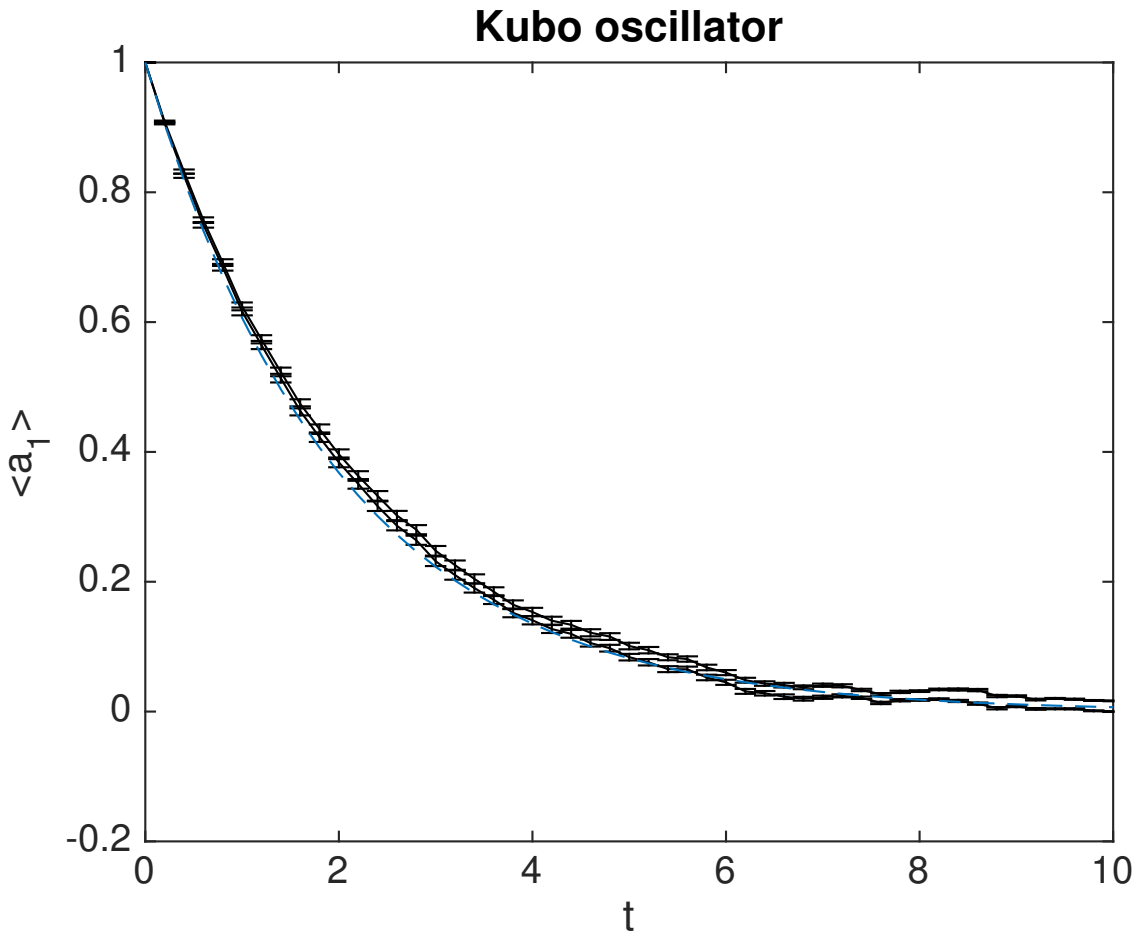

Fig. 3.2: The amplitude decay of a Kubo oscillator.

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, say `in.file = `filename.mat``.
- Run the simulation with *xsim()*.
- Run `xgraph(`filename.mat`)`, 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 ~`. 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 filename is still available in an interactive session workspace, just type

```
xgraph(in.file)
```

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

More generally, one can use any file name directly in *xgraph()*, which works with either matlab or HDF5 file types. 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. In the current version, in order to access the function handles in the saved files, Matlab needs to access the original input file. Hence, when you move the data to a new computer, it is best to include the original input file in the same directory as the data, to make the handles available.

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 change the graphics inputs in the `in` structure, then use the standard graphics command:

```
xgraph(data, in)
```

To change cell contents for a sequence, be aware that sequence inputs are stored in cell arrays with curly bracket indices, so you have to change them individually using an index.

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)
```

In both 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.

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:

```
in1.name = 'My first simulation'
in2.name = 'My next simulation'
```

This method requires that the `data`, `in1` and `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!

CHAPTER

FOUR

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 *checks* to 1. 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.

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

4.1 Wiener process

Try increasing the time resolution and adding a heading to the random walk example in *Interactive xSPDE*.

This requires specifying the number of points using *points*. To name the simulation, use *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`.

To run the xSPDE program after adding these inputs, click 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?

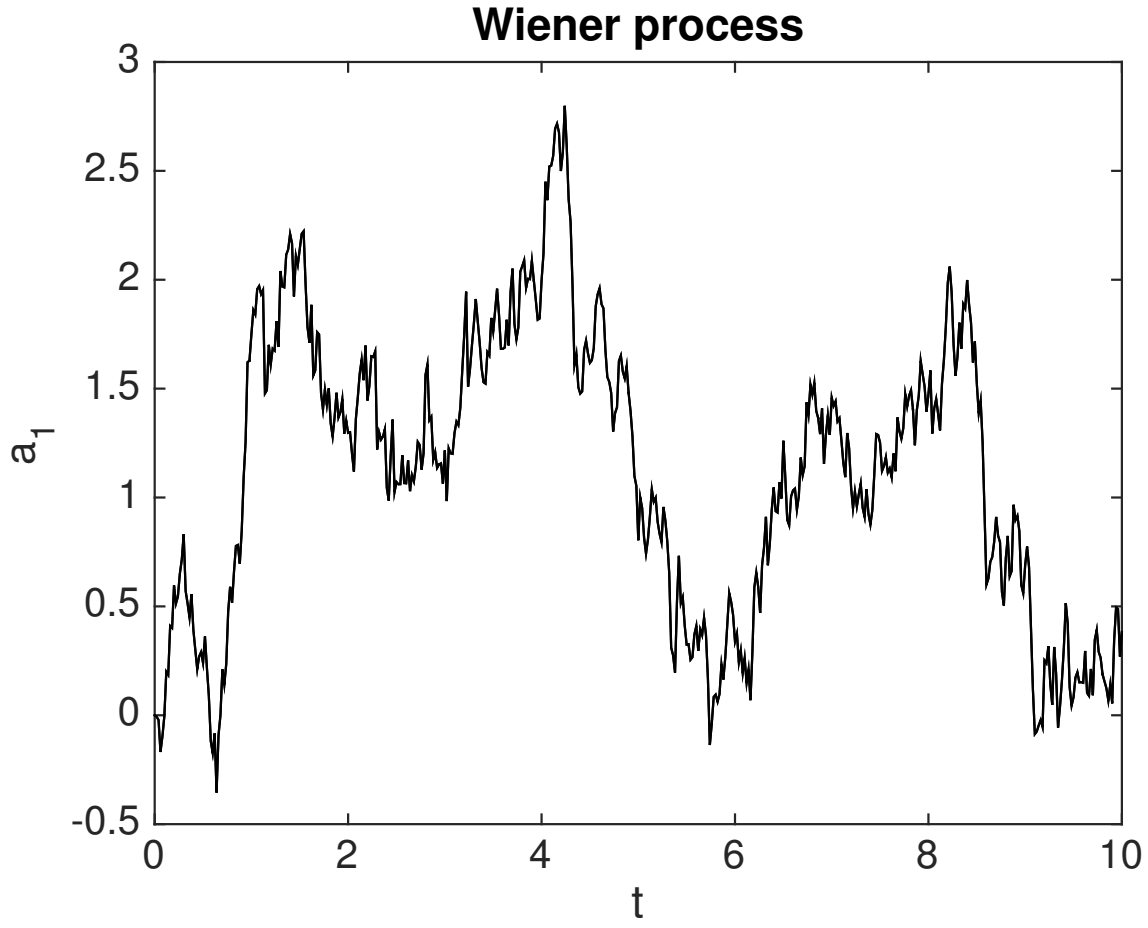

Fig. 4.1: Increasing the Wiener process resolution and adding a header.

4.2 Harmonic oscillator

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

$$a(0) = 1 + v,$$

where

$$\langle v^2 \rangle = 1$$

and the differential equation:

$$\dot{a} = ia + w(t).$$

4.2.1 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:

```
clear
in.initial = @(rv,~) 1+rv;
in.ensembles = [20,20];
in.da = @(a,w,~) i*a + w;
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:

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 an error summary, using the default number of points (51), for the sampling error and the step error.

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. The difference between the fine result and the coarse result gives the step error estimate, which is usually very conservative. Sampling errors are estimated from statistics of stochastic results only when the higher level ensembles are used.

4.2.2 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 (*compare*). All functions and data relating to observables are cell arrays, hence the curly brackets: `compare{1}` is the first element of an array of comparison functions that might be needed if there are many observables.

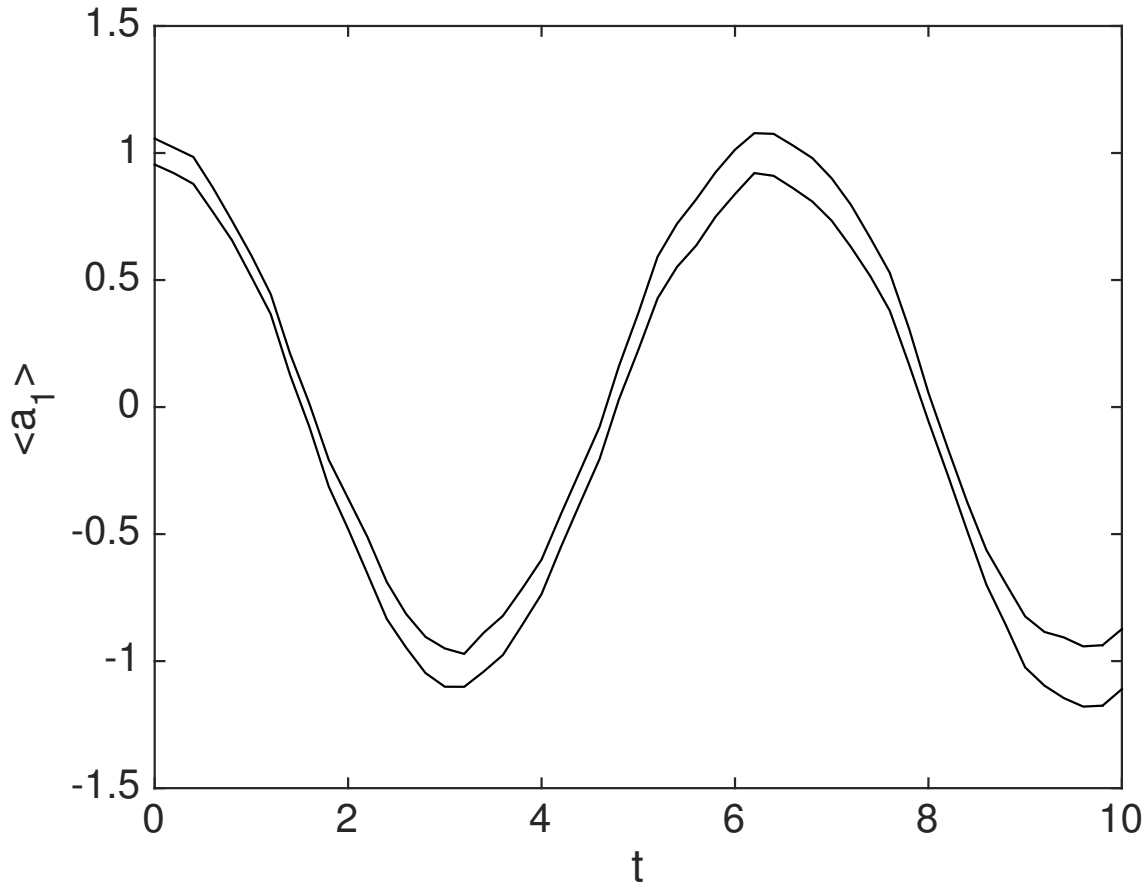

Fig. 4.2: Simple harmonic oscillator amplitude

```
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:

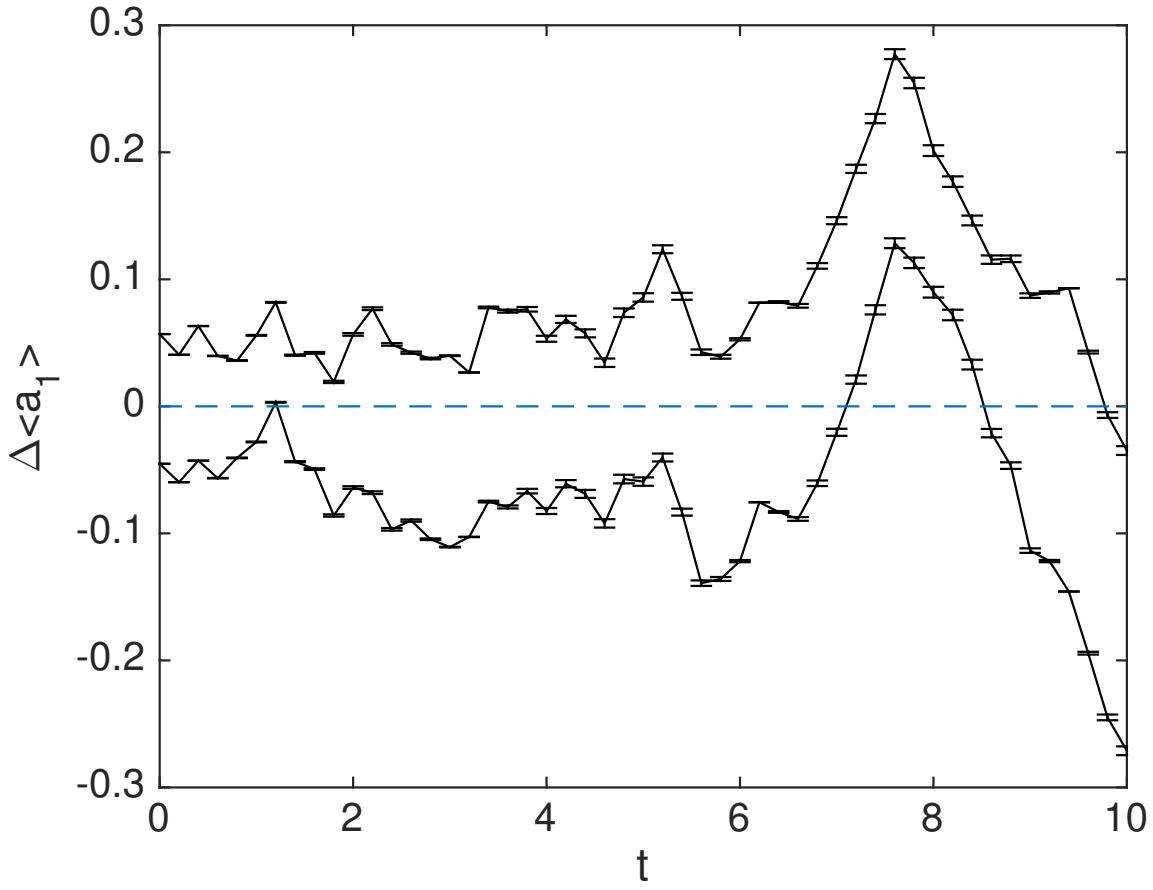

Fig. 4.3: Simple harmonic oscillator comparison graph: exact vs computed, with error-bars.

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?

4.3 Kubo oscillator

The next example is more interesting. It is the Kubo oscillator, an oscillator with a random frequency. It is a case of multiplicative noise, but with a complex variable.

In Stratonovich stochastic calculus, its equation is:

$$\dot{a} = ia\zeta(t)$$

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

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

where

$$w(t) = \int_0^t \zeta(\tau)d\tau$$

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

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

4.3.1 Kubo initial conditions and derivative

Here more parameters are needed. One real noise term is required per integration point, specified using *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 = @(rv,r) 1+0*rv;
    in.da = @(a,w,r) i*w.*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 = 1.065936e-02
-  Max step error = 5.072889e-04
-  Max comparison difference = 1.269069e-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.

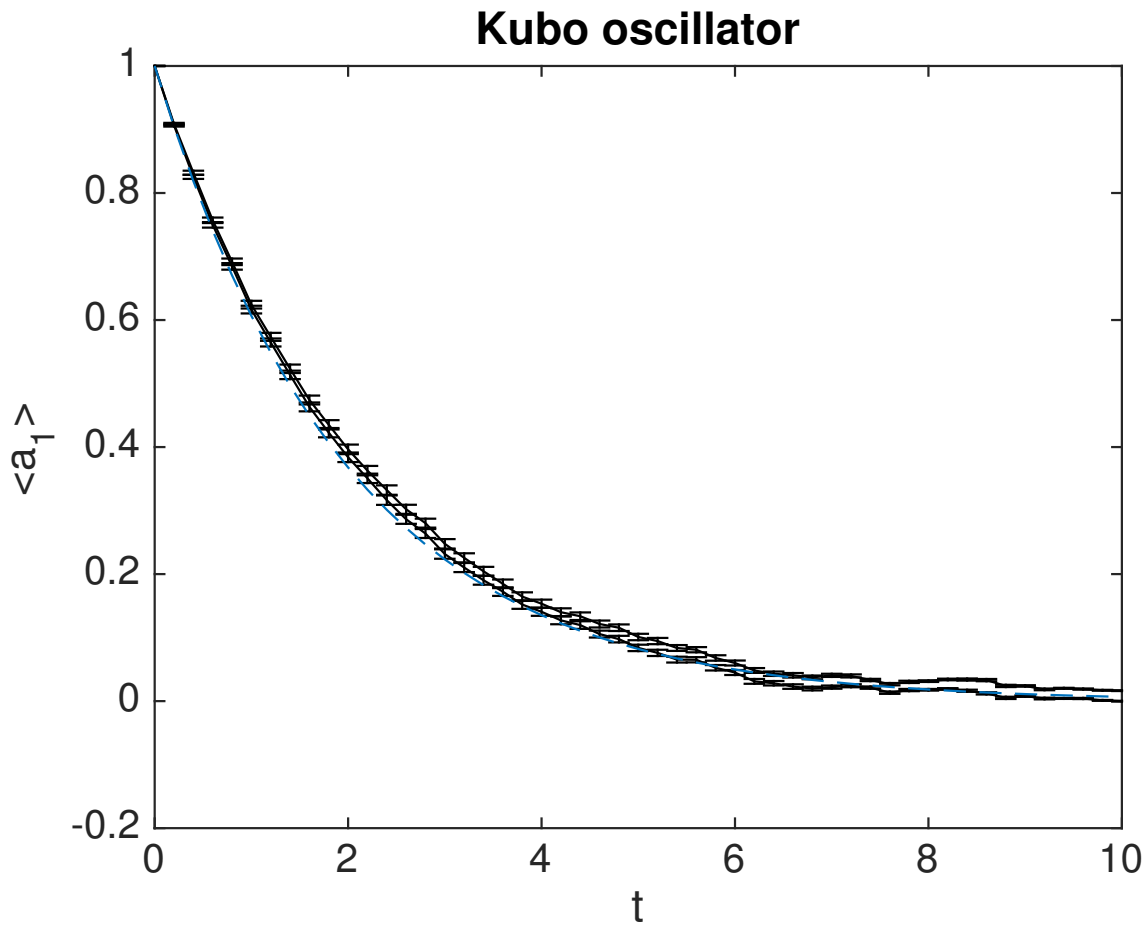

Fig. 4.4: Kubo oscillator mean amplitude

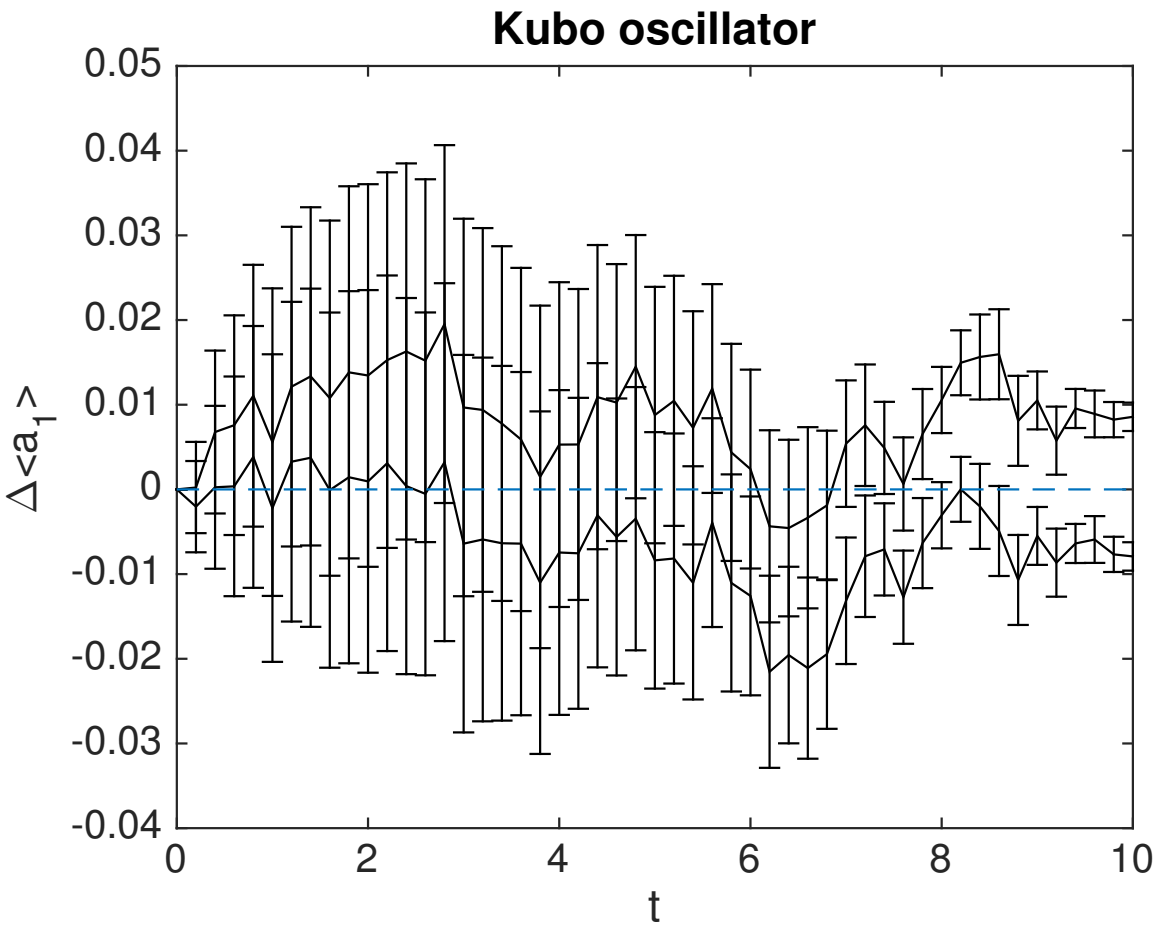

Fig. 4.5: Kubo oscillator amplitude errors

4.3.2 Kubo graphs

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

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.

4.4 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:

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

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

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

4.4.1 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 = @(r) 0.5*i*(r.Dx.^2-1.0);
    in.olabels = {'a_1(x)'};
    in.compare{1}= @(t,~) 1;
    e = xspde(in);
end
```

The output reflects the known analytic result.

4.4.2 Soliton graphs and errors

Graphs of results are given below.

The xgraph program reports that comparison errors are slightly less than the step error, but 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.

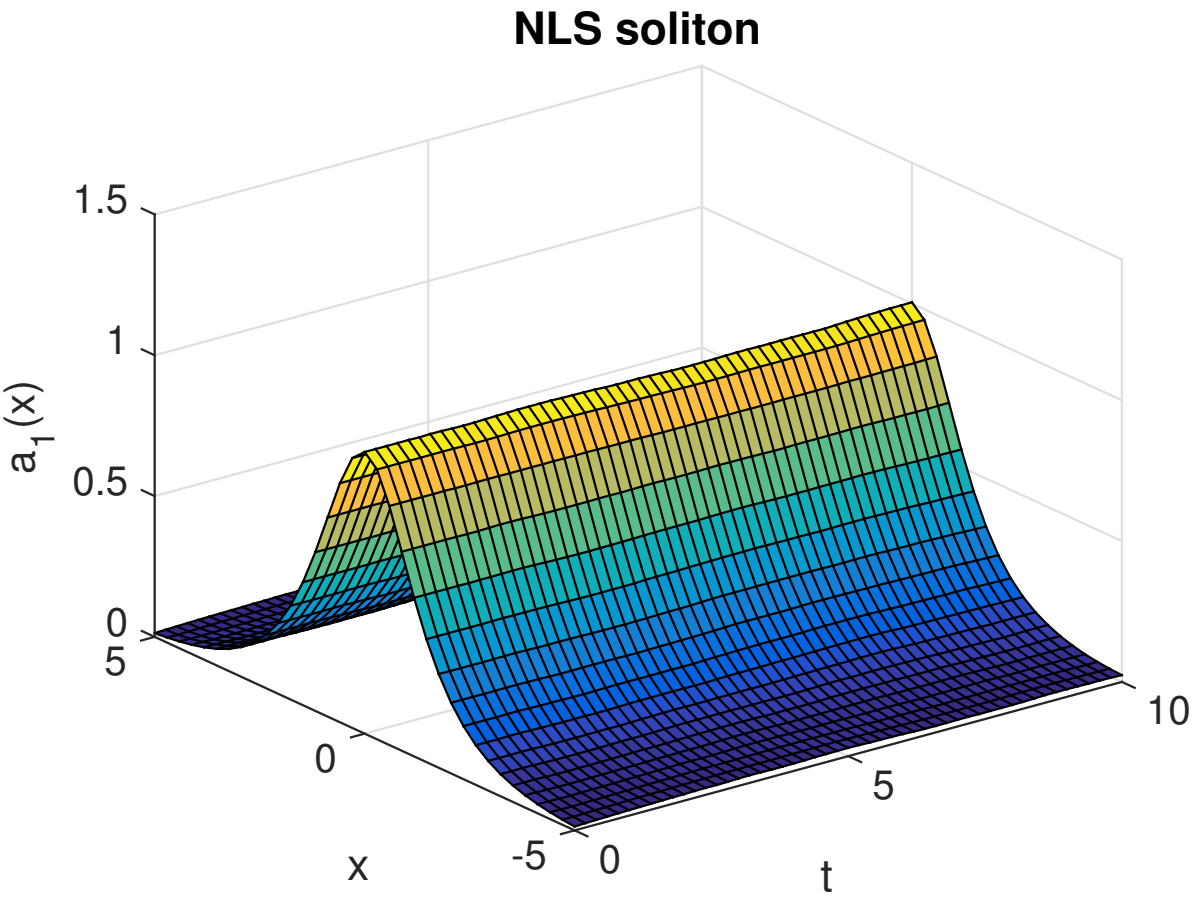

Fig. 4.6: Soliton amplitude versus space and time

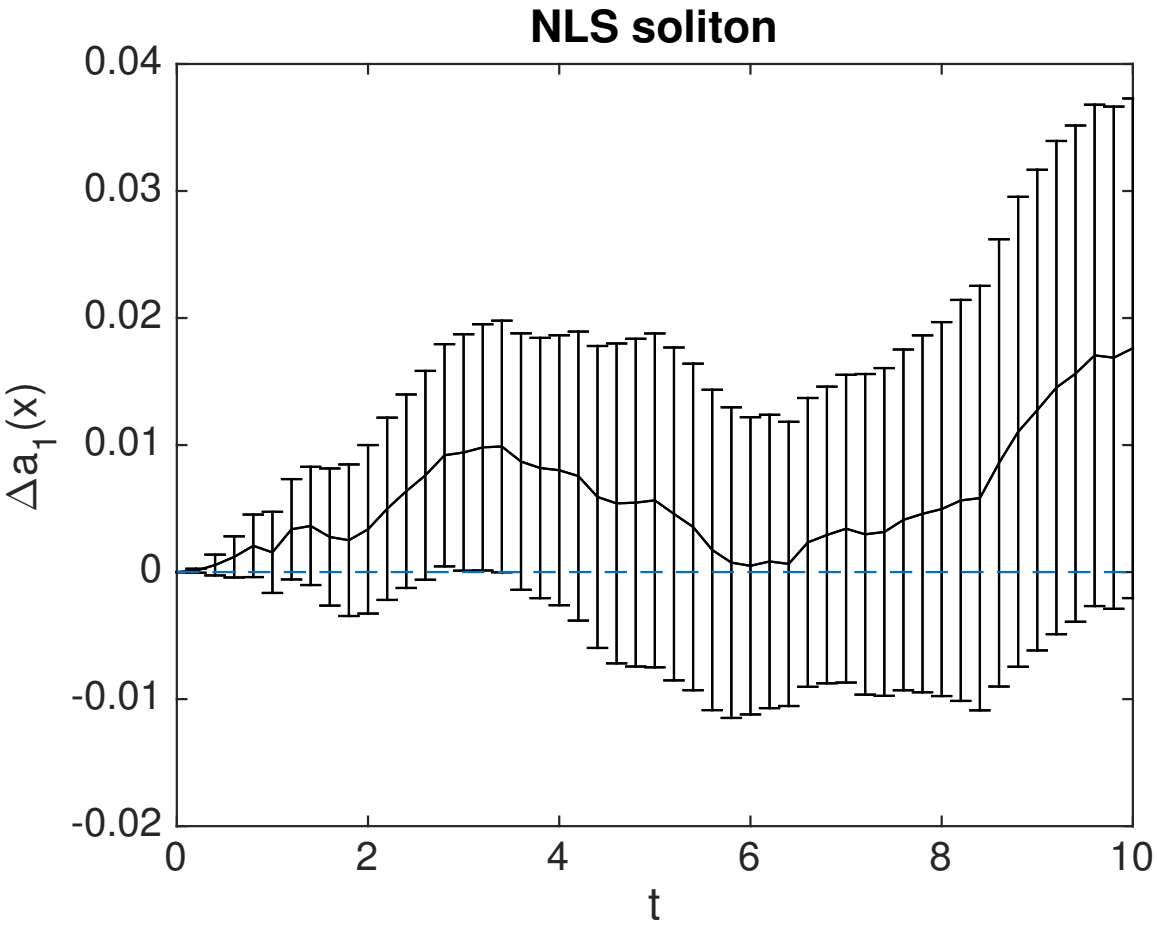

Fig. 4.7: Soliton amplitude errors at center

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.

4.5 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:

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

4.5.1 Gaussian inputs

Before running this simulation, be careful to change the Matlab working directory to your intended working directory, which must have write permission enabled. For example, type:

```
cd ~
```

A possible user set of parameters to simulate this is:

```
function [e] = Gaussian()
    in.dimension = 4;
    in.initial = @(~,r) exp(-0.5*(r.x.^2+r.y.^2+r.z.^2));
    in.da = @(a,~,~) zeros(size(a));
    in.linear = @(r) 1i*0.05*(r.Dx.^2+r.Dy.^2+r.Dz.^2);
    in.observe = {@(a,~) a.*conj(a)};
    in.olabels = '|a(x)|^2';
    in.file = 'Gaussian.h5';
    in.images = 4;
    in.imagetype = 1;
    in.transverse = 2;
    in.headers = 1;
    in.compare{1} = @(t,~) [1+(t/10).^2].^(-3/2);
    [e,in] = xsim(in);
    e = e+xgraph(in.file);
end
```

Here the program writes an HDF5 data file using *xsim()*, and then reads it in with the stored file-name, using *xgraph()*. Note that *xsim()* may have to change the file-name to avoid overwriting any old data. In this case, it returns the new file-name is uses. The program reports the following maximum step-size errors, which in this case are negligible, as they are around $\sim 10^{-15}$, and are purely due to the interaction picture transformations. These errors do not depend on step-size, apart from rounding.

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

4.5.2 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.

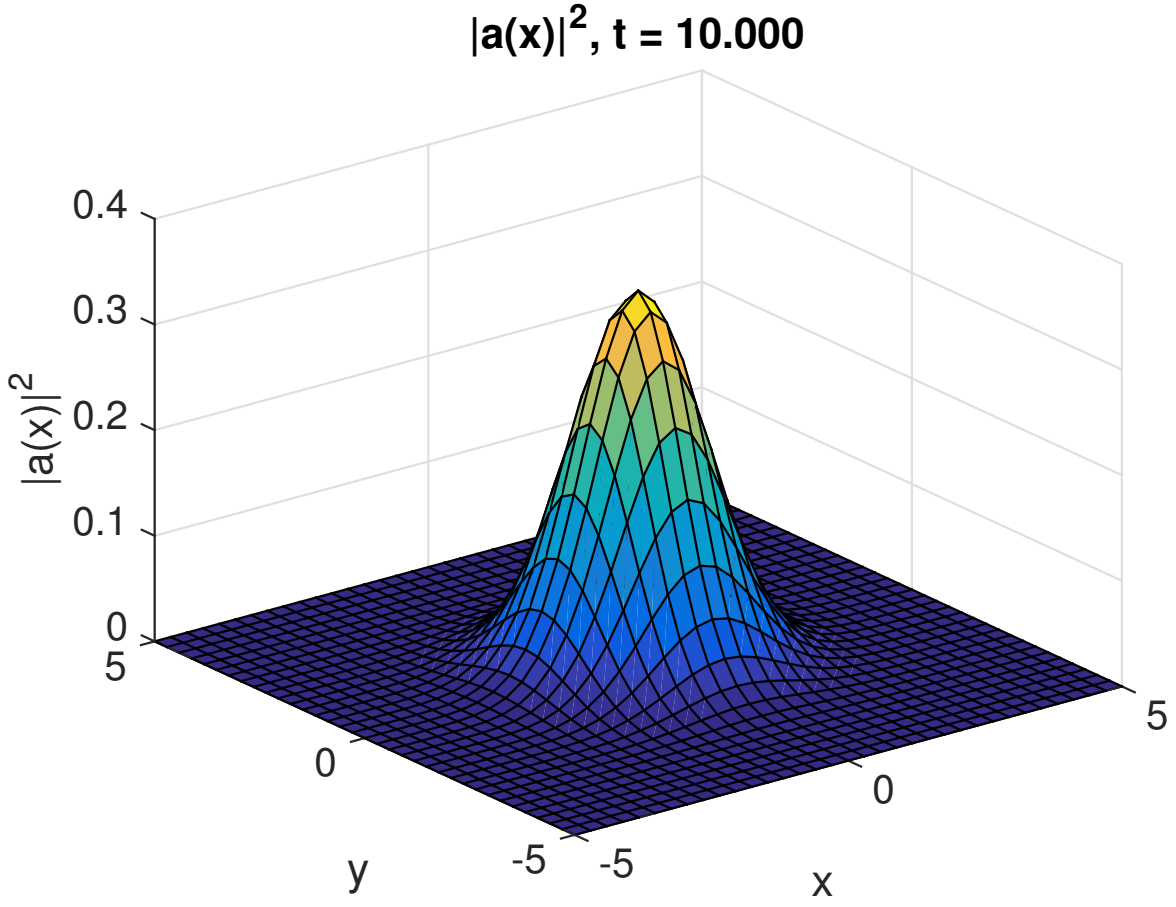

Fig. 4.8: Image of transverse gaussian intensity at $t = 0$.

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.

4.6 Planar noise

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

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

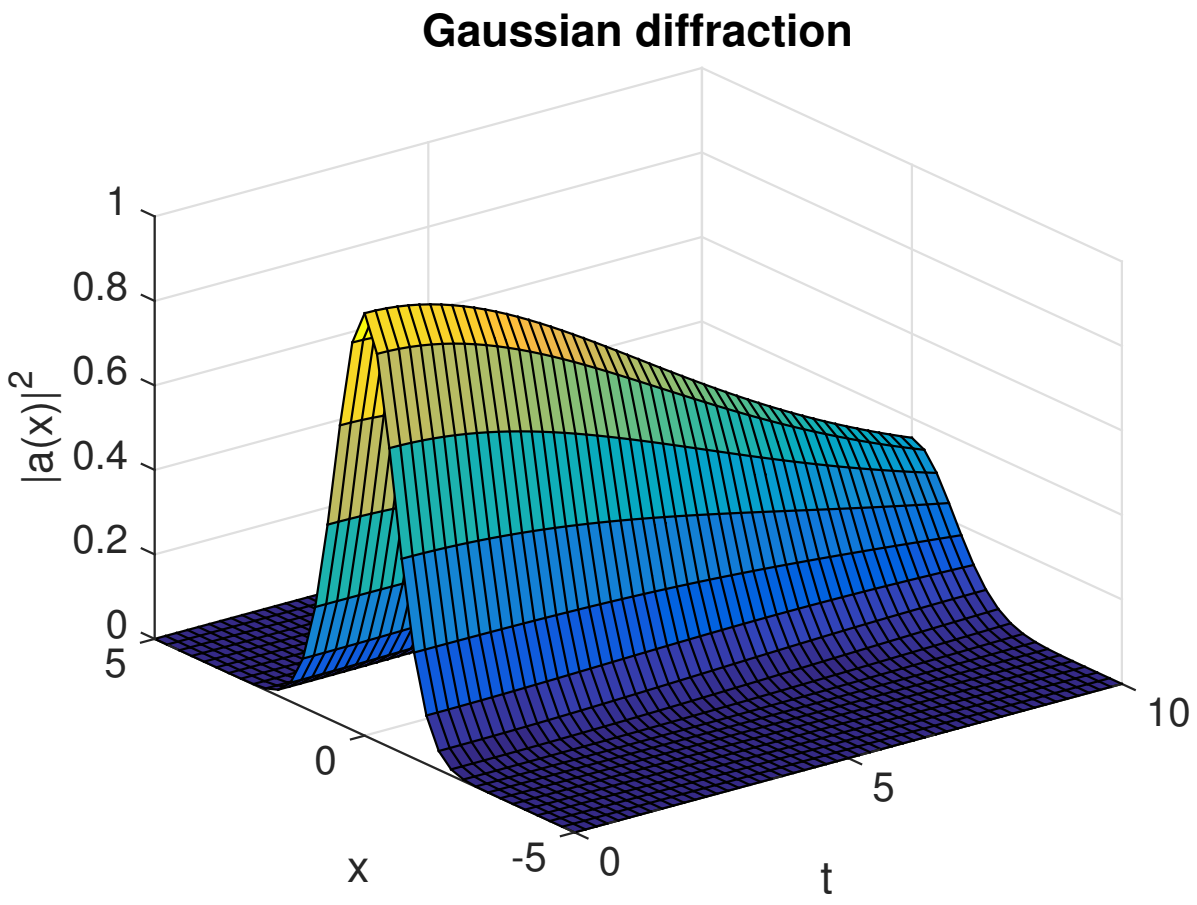

Fig. 4.9: Gaussian intensity diffraction

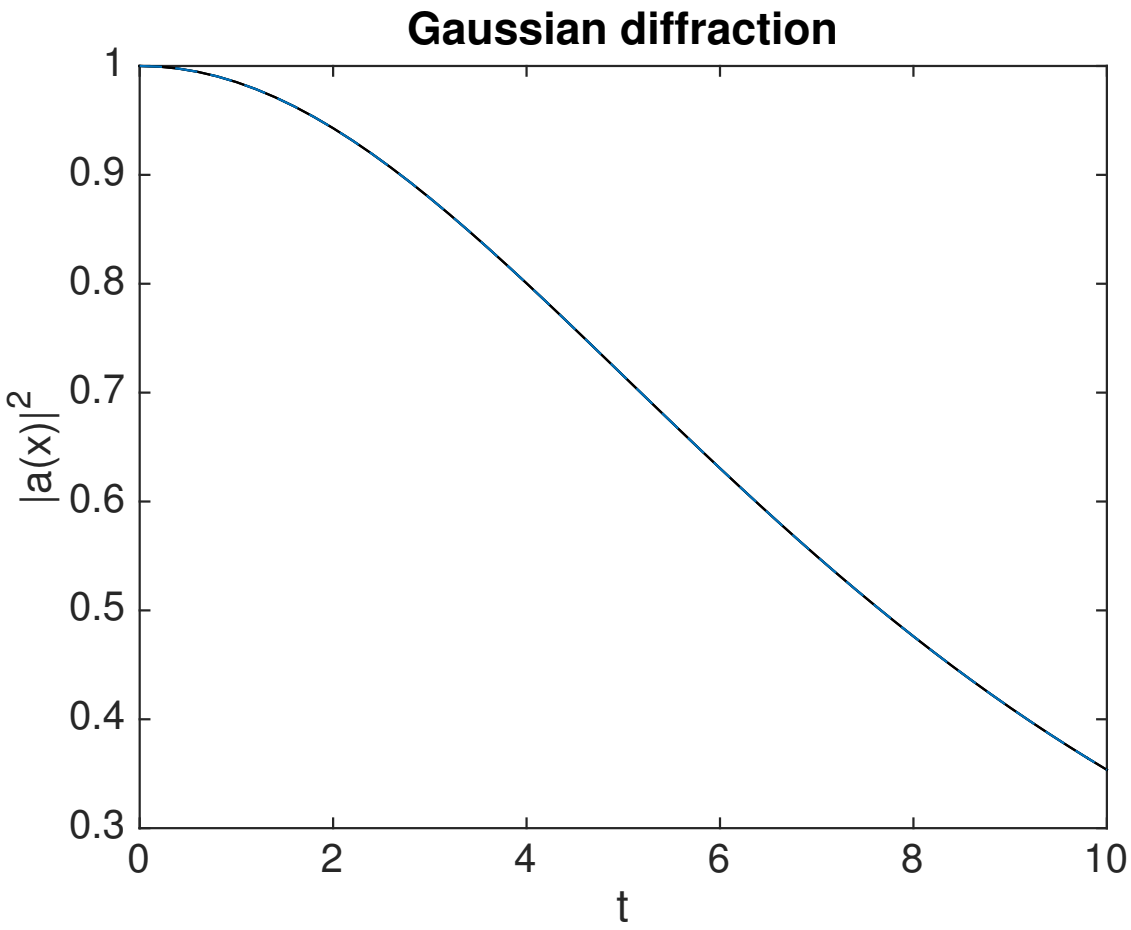

Fig. 4.10: Gaussian intensity at $\boldsymbol{r} = 0$.

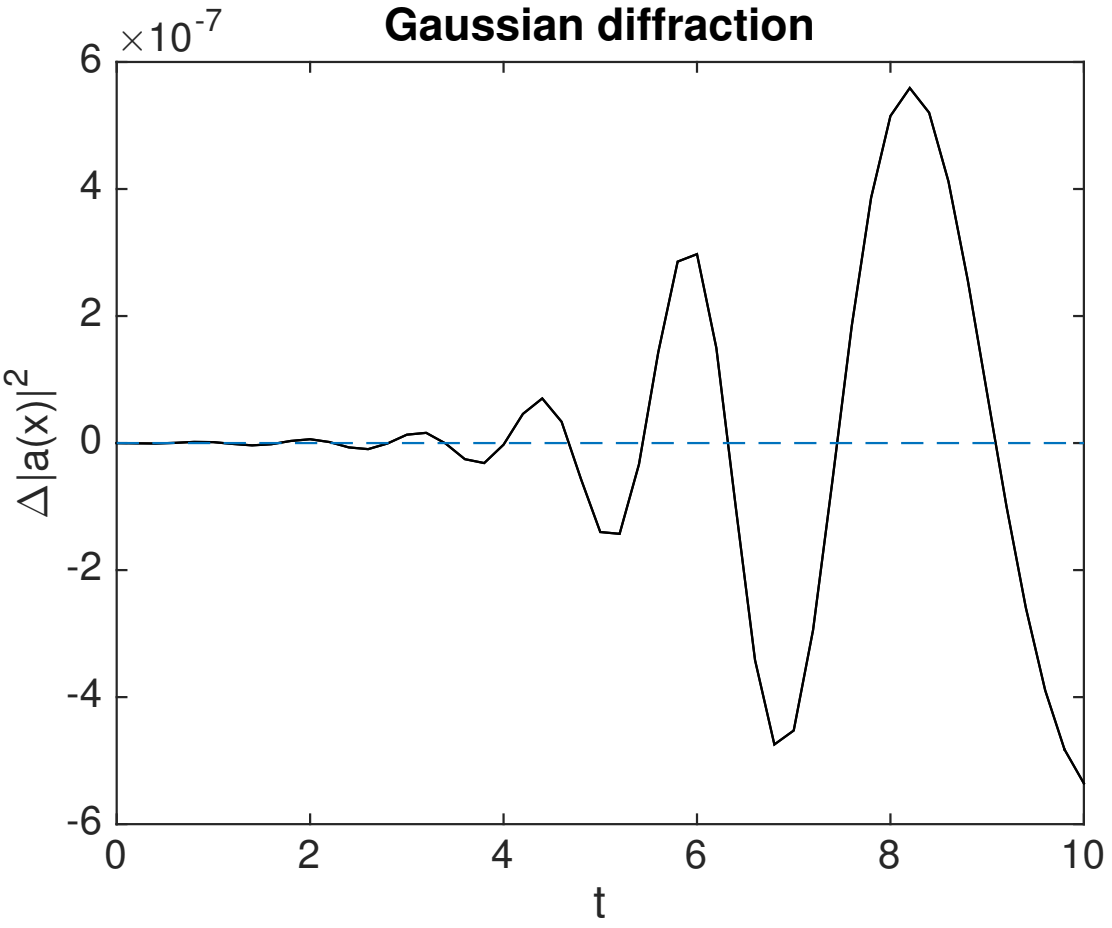

Fig. 4.11: Gaussian, modulus-squared errors at $\boldsymbol{r} = 0$.

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:

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

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:

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

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 uses an internal N-dimensional inverse FFT before being added, which is slower, so this method is not recommended unless needed.

4.6.1 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 = [10,2,2];
    in.initial = @Initial;
    in.da = @D_planar;
    in.linear = @Linear;
    in.observe{1} = @(a,r) xint(a(1,:).*conj(a(1,:)),r);
    in.observe{2} = @(a,r) xint(a(2,:).*conj(a(2,:)),r.dk,r);
    in.observe{3} = @(a,r) xave(a(1,:).*conj(a(2,:)));
    in.transforms = {[0,0,0],[0,1,1],[0,1,1]};
    in.olabels{1} = '<\int|a_1(x)|^2 d^2x>';
    in.olabels{2} = '<\int|a_2(k)|^2 d^2k>';
    in.olabels{3} = '<<a_1(k)a^*_2(k)>>';
    in.compare{1} = @(t,in) [1+t]*in.nspace;
    in.compare{2} = @(t,in) [1+t]*in.nspace;
```

(continues on next page)

(continued from previous page)

```
    in.compare{3} = @(t,in) 0;
    in.images = {4,2,0};
    in.transverse = {2,2,0};
    in.pdimension = {4,1,1};
    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 = D_planar(a,w,r)
    da(1,:)  = (w(1,:)+1i*w(2,:))/sqrt(2);
    da(2,:)  = (w(3,:)+1i*w(4,:))/sqrt(2);
    end
function L = Linear(r)
    lap = r.Dx.^2+r.Dy.^2;
    L(1,:)  = 1i*0.5*lap(:);
    L(2,:)  = 1i*0.5*lap(:);
end
```

4.6.2 Planar graphs

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

Exercise

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

4.7 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, g3m ...}`. 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:

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

4.7.1 Input file

The full input file is given below.

```
function [e] = Gain()
    in.name = 'Loss with noise';
    in.ranges = 4;
```

(continues on next page)

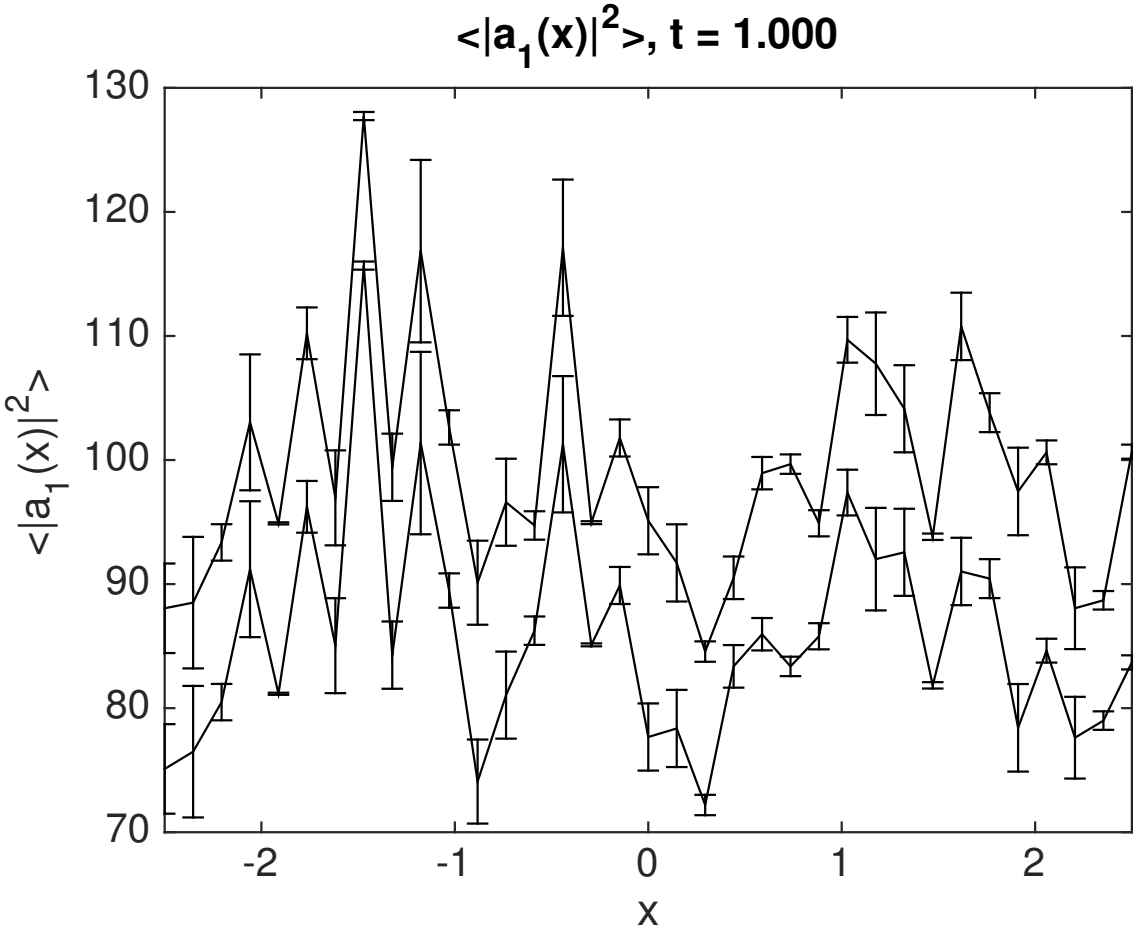

Fig. 4.12: Planar noise intensity as a transverse slice in the $t = 1$, $y = 0$ plane. The relatively large sampling error is because there are not many samples.

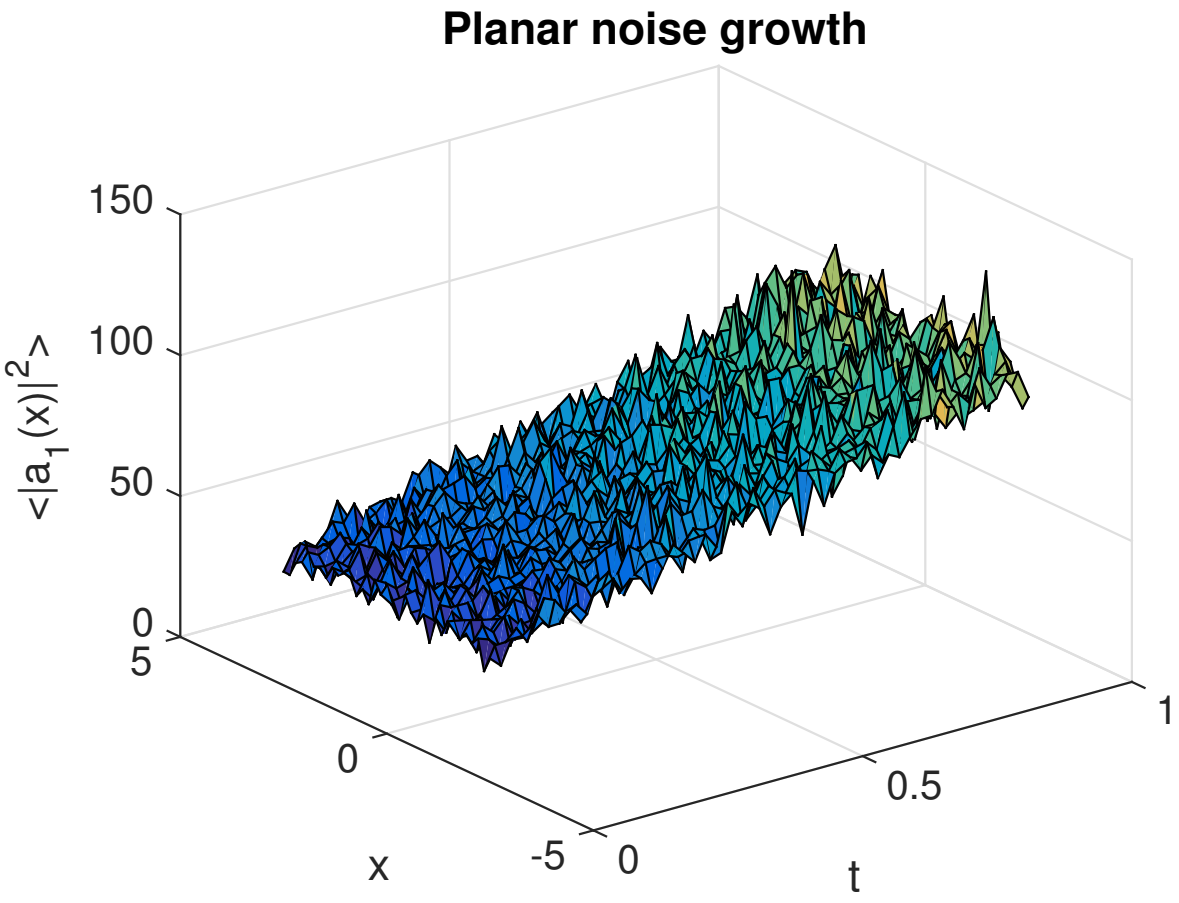

Fig. 4.13: Growth in noise intensity with time vs. x, at $y = 0$.

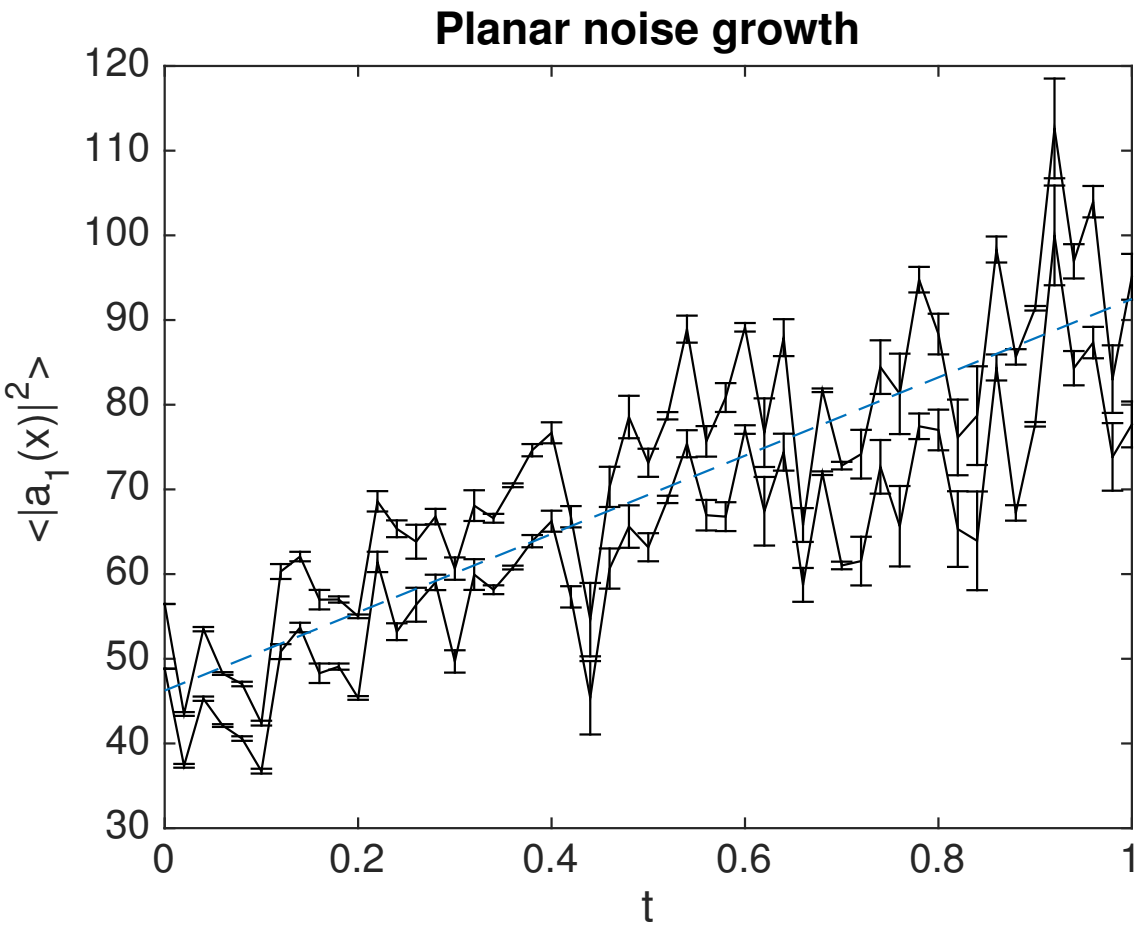

Fig. 4.14: Growth in planar noise intensity at $x = y = 0$, vs. exact results.

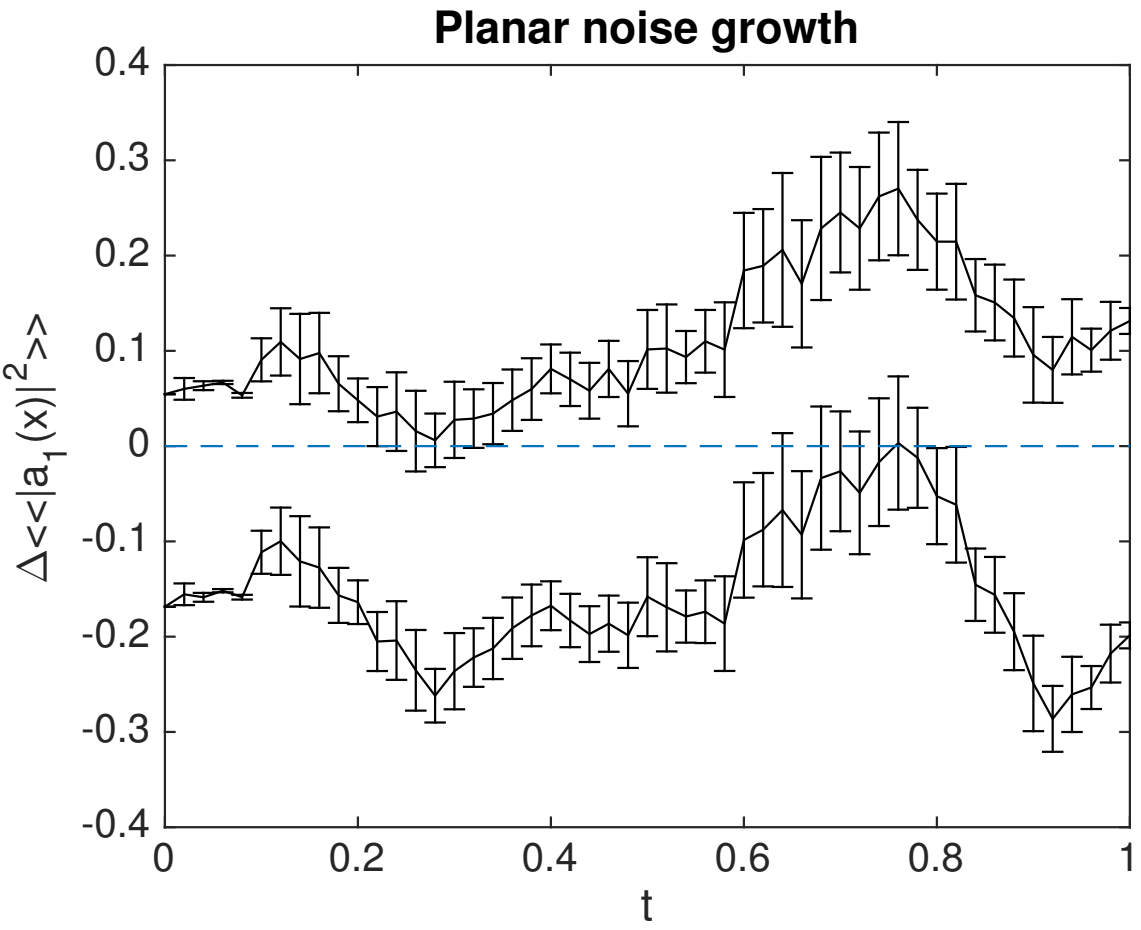

Fig. 4.15: Errors in planar noise intensity at $x = y = 0$, vs. exact results. These results are averaged across the plane, as well as being ensemble averaged.

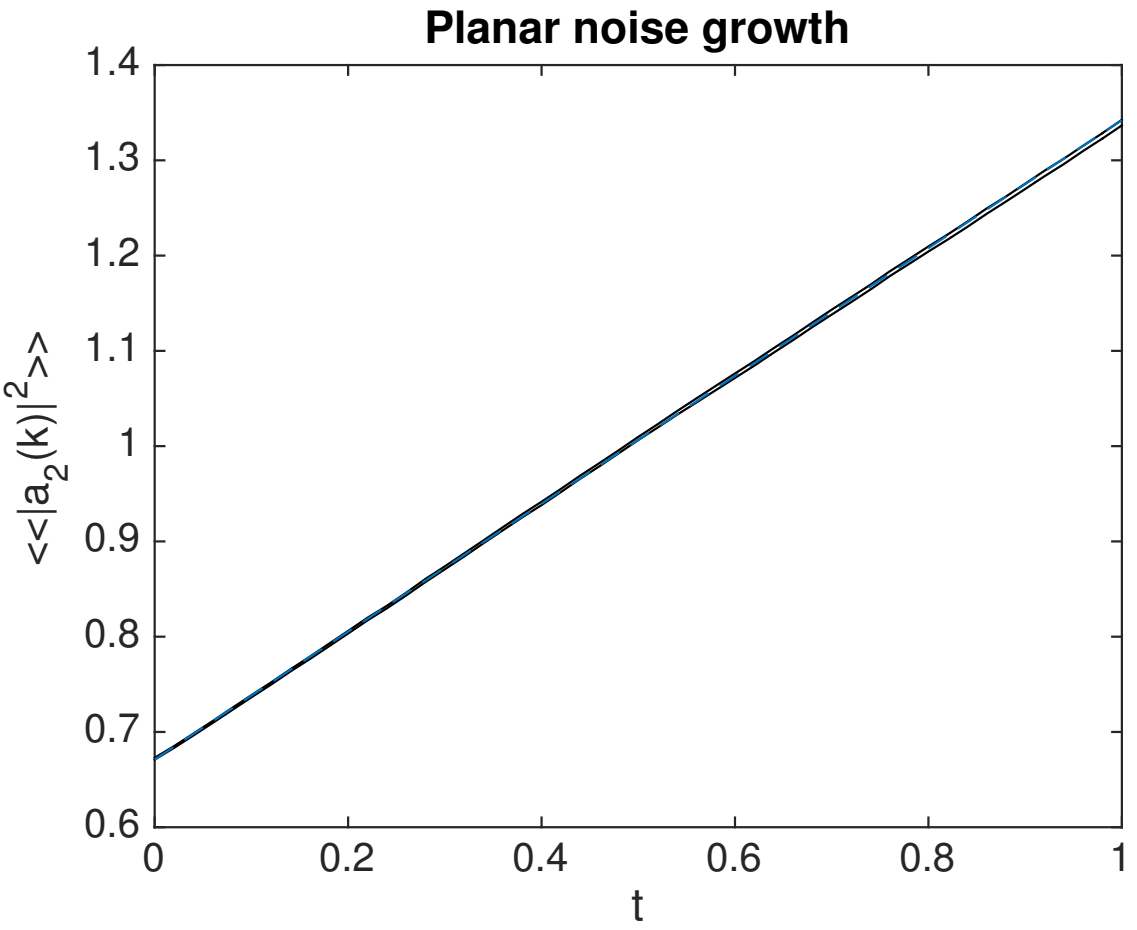

Fig. 4.16: Growth in planar noise intensity in momentum space, for the second field, at $k_x = k_y = 0$.

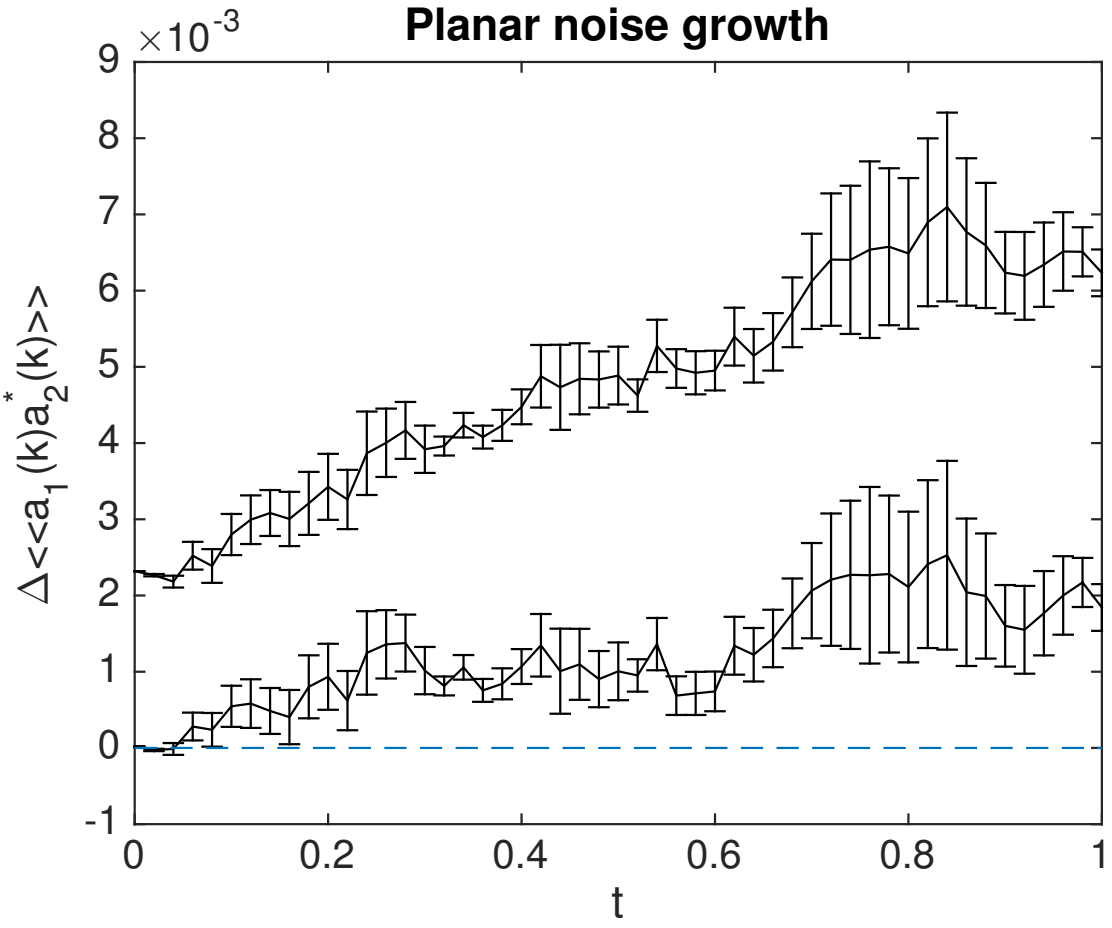

Fig. 4.17: Lattice averaged errors in cross-correlations in momentum space, vs. exact results.

(continued from previous page)

```
    in.noises = [2,0];
    in.ensembles = [100,16,1];
    in.initial = @(v,~) (v(1,:)+1i*v(2,:))/sqrt(2);
    in.da = @(a,w,r) -a + w(1,:)+1i*w(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` 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.

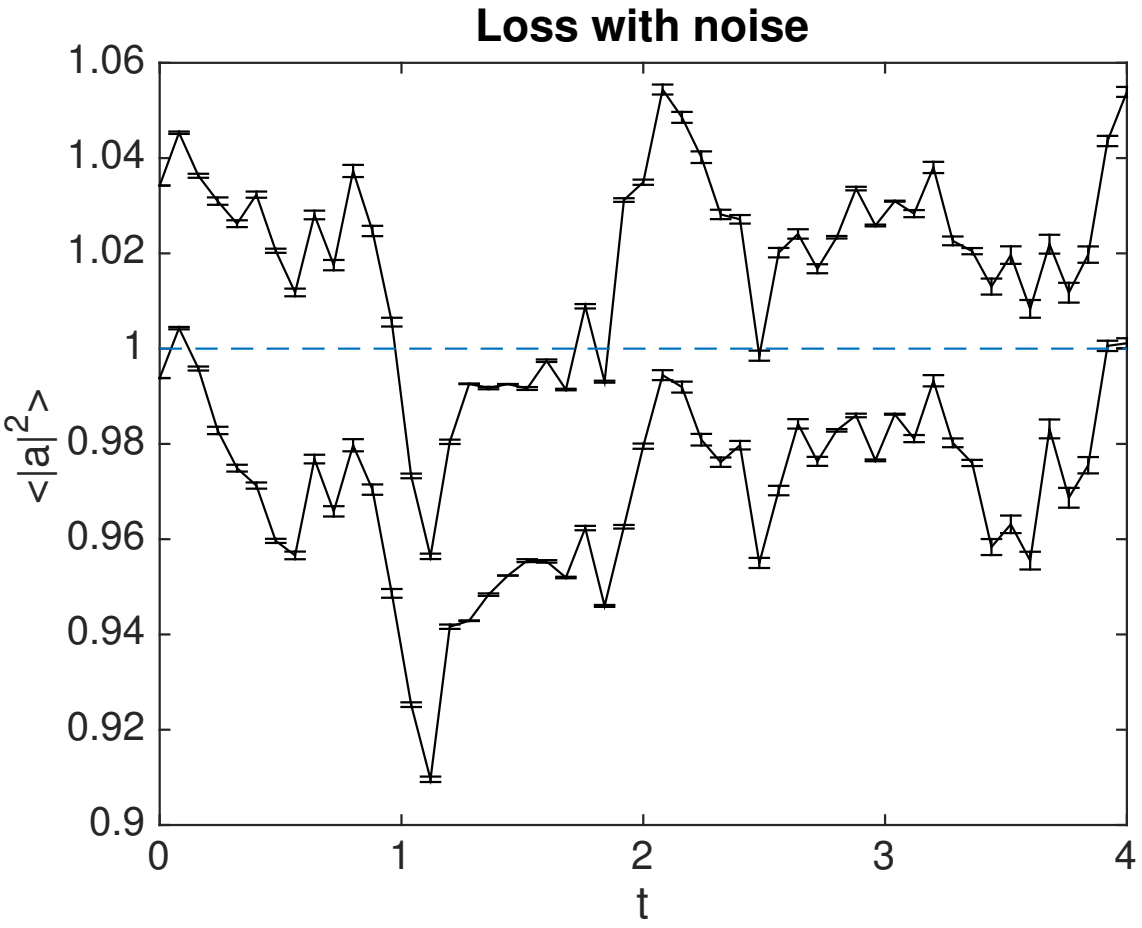

Fig. 4.18: Absorber intensity

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

4.7.2 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 `compare` functions in the project file. The time axis in the second graph has the origin reset to zero.

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

Exercise

Reverse the order of gain and loss.

4.8 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:

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

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

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

4.8.1 Characteristic inputs

The important parameters and functions in this case are:

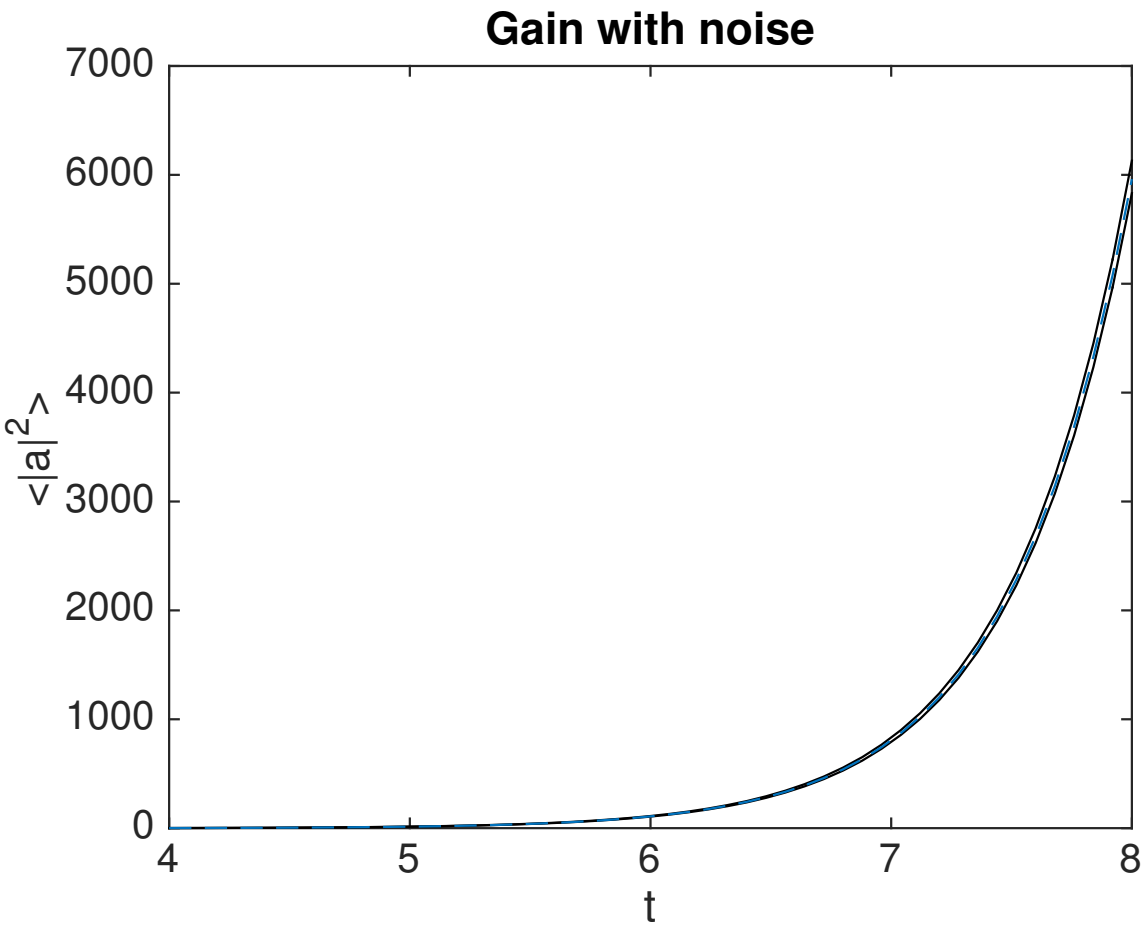

Fig. 4.19: Noisy amplifier intensity

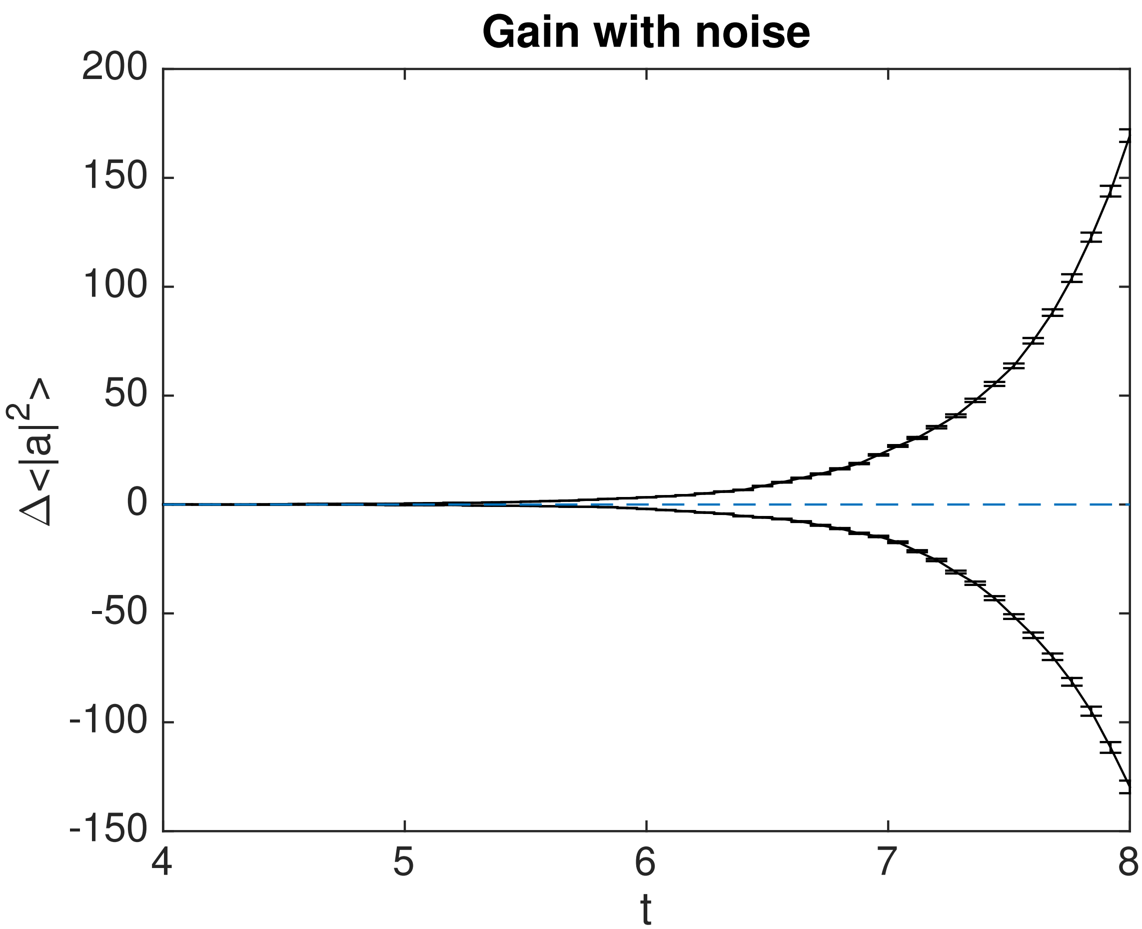

Fig. 4.20: Noisy amplifier intensity errors, showing how the sampling errors increase in time.

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

The simulation program reports step errors of order of the intrinsic rounding error, which is slightly misleading, since while the interaction picture is essentially exact, it is solving a finite lattice problem exactly. This transverse lattice discretization does introduce transverse discretization errors in addition, and these are seen from the comparisons with the exact results. The lesson to be learnt here is that one must check the transverse discretization errors in addition, by changing the transverse lattice.

Graphs of results are given below.

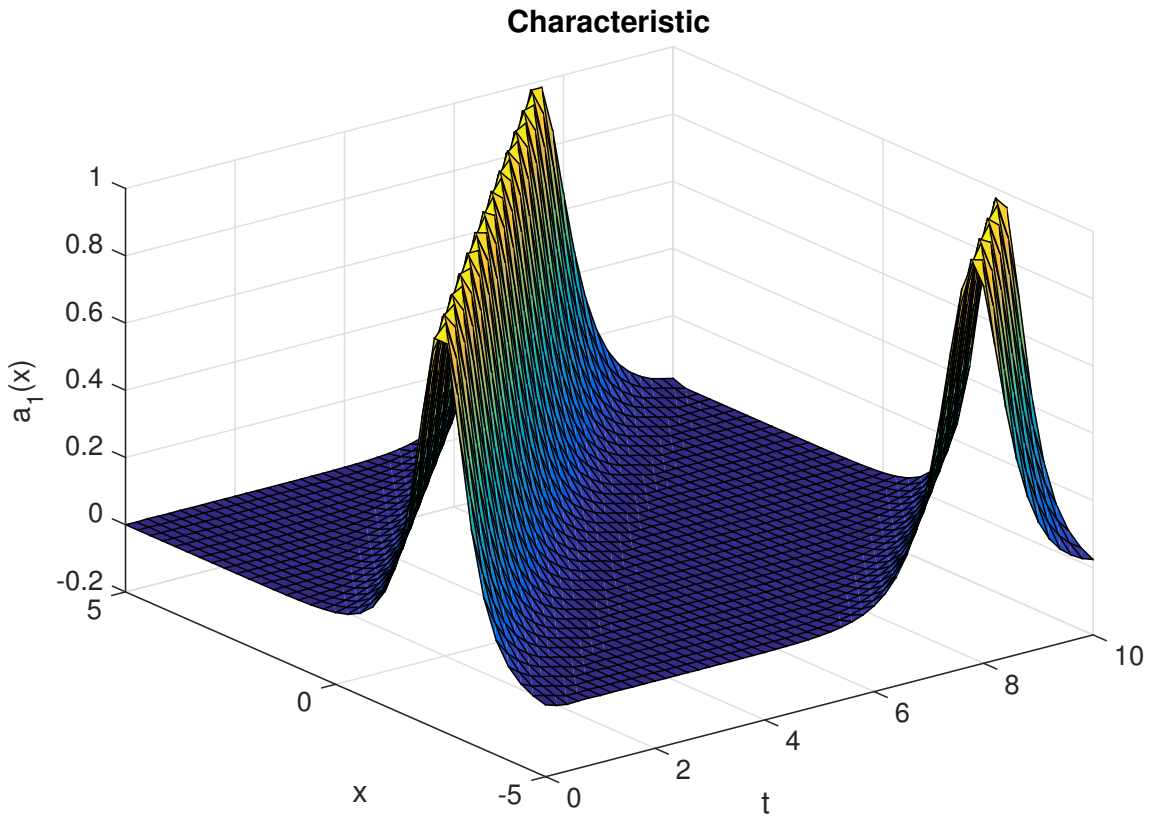

Fig. 4.21: Characteristic traveling wave versus space and time

Exercise

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

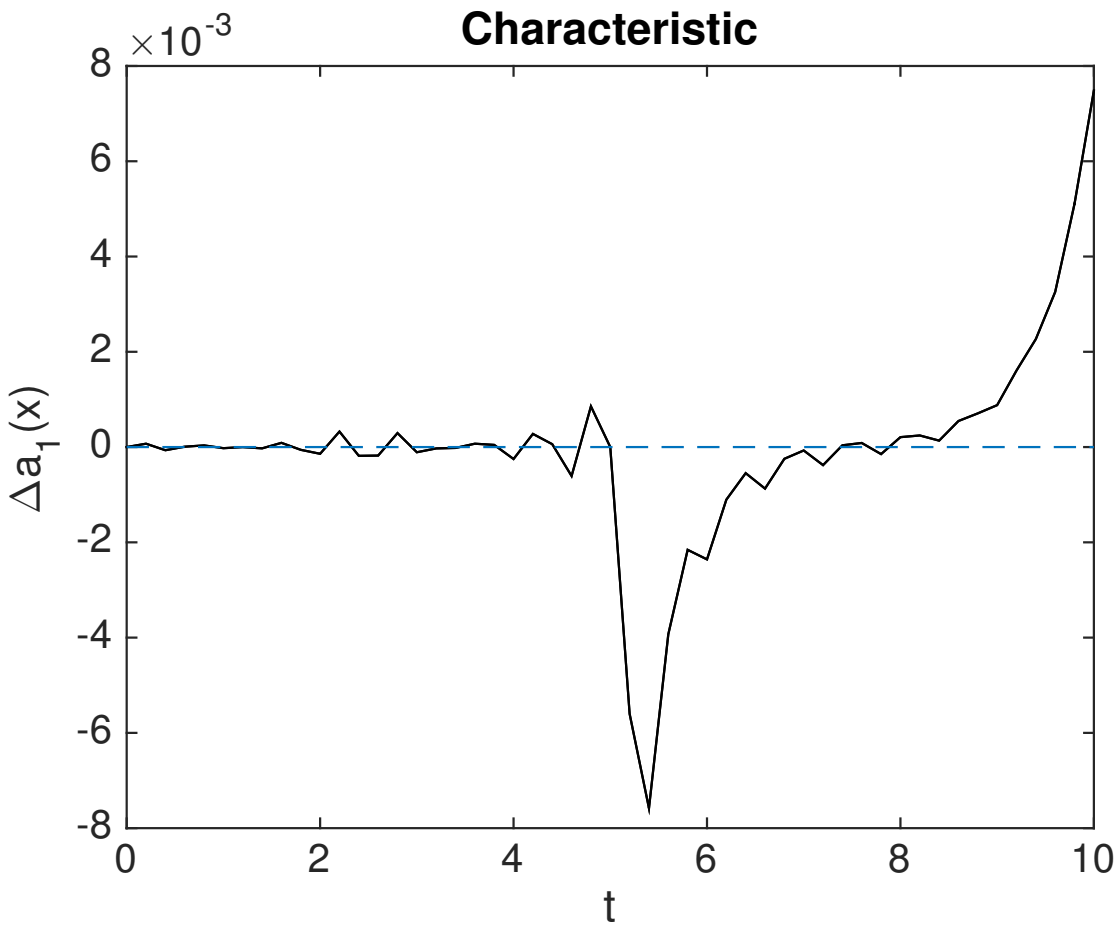

Fig. 4.22: Characteristic errors at center

4.9 Equilibrium

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

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

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:

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

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

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

4.9.1 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 = [100,1,10];
    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.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,~)100.16./(pi*(1+w.^2))};
    e = xspde(in);
end
```

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

The xsim program will report in the error summary that the comparison differences indicate that the maximum error reported is typically about 1.5 standard deviations of the maximum sampling error. Given the number of data points, this is a reasonable result: statistical errors can exceed one standard deviation.

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.

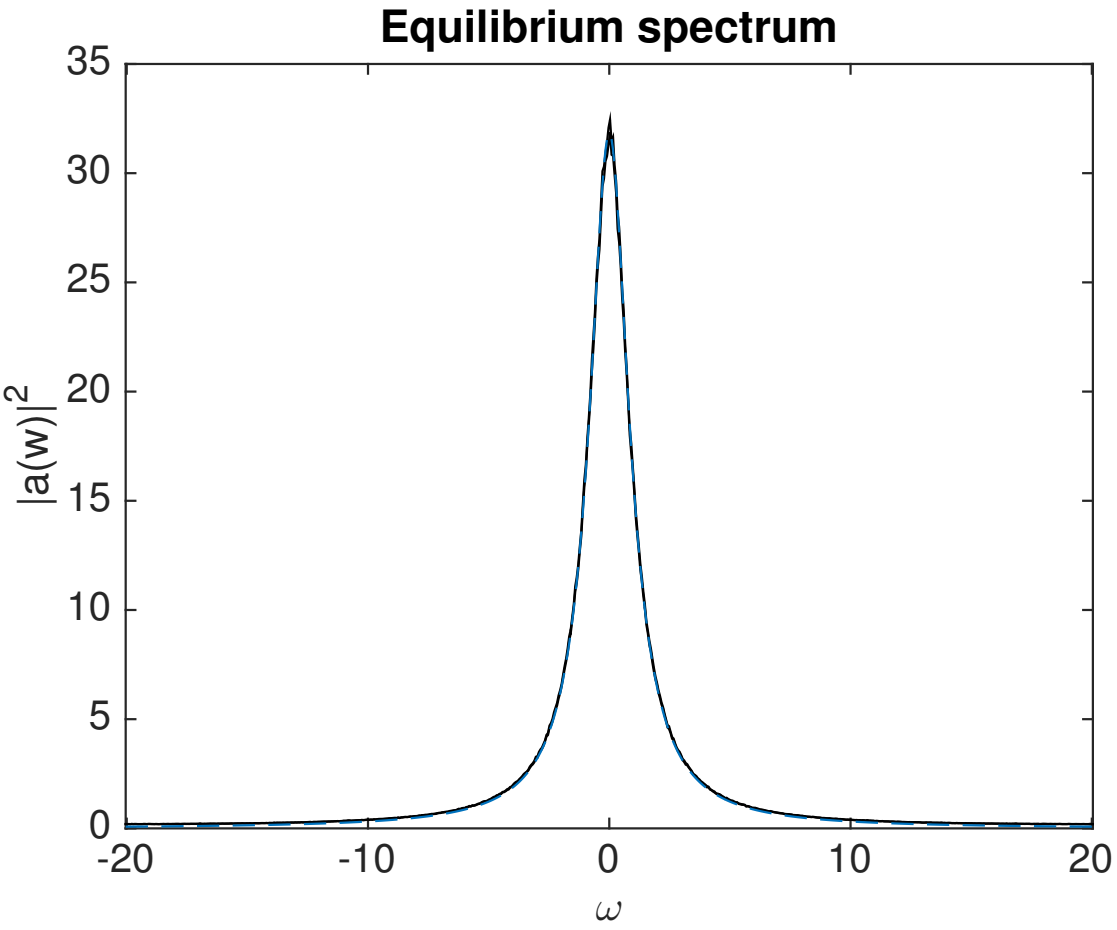

Fig. 4.23: Equilibrium spectral intensity

CHAPTER

FIVE

XSIM

The simulation program logic is straightforward. The main code 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. In this chapter, we go into the workings of the simulation program, xsim.

5.1 How xSIM works

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

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

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 a built-in multidimensional graphics tool, xGRAPH, treated in the next chapter.

5.1.1 Sequences

In many types of application, a sequence of stochastic equations requires simulation. In these cases the final field value after integration of one equation becomes the initial value of the next equation in the sequence. At the end of each simulation loop, global averages and error-bars are calculated and stored for output.

Sequences therefore are the basic concept used in both the input of parameters to xSPDE, and the storage of data generated.

5.1.2 Input and data arrays

To explain xSIM 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 fields are the input of the next.
- The main function is called using `[error, input, data, rawdata] = xsim([rawdata,] 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. All outputs can be saved to disk storage if required.

The optional [data,] input is only used when there is previous raw data that needs analysis. If this is present, no new simulation takes place. Any `observe` functions in the new `input` will be employed to take further averages over the existing raw data. This allows re-analysis of large simulation data-sets without more simulations.

The returned `error` is a vector: the first component is the maximum error found, the second component is the elapsed time. The returned `input` structure is available to the user to give the data file-name, in case xSIM needs to store data with a new file-name. For data security, it will not overwrite existing data.

If xSIM is called within xSPDE, it will generate graphs with its own graphics program xGRAPH. Otherwise, data can be stored then graphed later using xGRAPH.

5.1.3 Customization options

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

Customization options include functions the allow user definition of:

- inputs
- interfaces
- stochastic equations
- boundary values
- 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.

User-defined functions have to return specified array sizes, compatible with the internal arrays in xSIM. These sizes are checked by xSIM prior to a simulation. The xSIM program will print out a record of its progress.

5.2 Averaged data

5.2.1 Observables and functions

To allow options for taking averages, these are carried out in two stages. The first type of average is a local average, taken over any function of the locally stored ensemble of trajectories. These use the :func:observe functions, specified by the user. The default is the real values of each of the fields, stored as a vector. Multiple observe functions can be used, and they are defined as a cell array of functions.

Next, any function can be taken of these local averages, using the :func:function transformations, again specified by the user. The default is the original set of local averages. This is useful if different combinations, such as normalised

ratios are needed, or to combine the averages at different times. These second level function outputs are then averaged again over a second level of ensemble averaging, if specified. This is used to obtain estimates of sampling and step-size errors in the final data outputs.

This is explained below in more detail.

5.2.2 Observe functions

During the calculation, observables are calculated and averaged over the `ensembles(1)` parallel trajectories in the *`xpath()`* function. These are determined by the functions in the *`observe()`* cell array.

The number of *`observe()`* functions may be smaller or larger than the number of vector fields. The observable may be a scalar or vector. These include the averages over the ensembles, and can be visualized as a single graph with one or more lines. The *`observe()`* functions use for input and output the flat or matrix type internal arrays.

Next, arbitrary functional transforms can be taken, using the *`function`* cell array. These functions can use as their input the full set of *`observe()`* output data cell arrays, including a time index. They default to the original *`observe()`* data if they are not user-defined. Functional transforms are most useful if one wishes to use functions which require knowledge of normalization or ensemble averages of lower-level data. There can be more *`function`* definitions than *`observe()`* functions if needed.

Each *`observe()`* function or transformation in *`xsim()`* defines a single logical `graph` for the simulation output. However, the graphics function *`xgraph()`* can generate several projections or views of the same dataset, as explained below.

5.2.3 Combined observables: `data`

These results are added to the earlier results in the cell array `data`, to create a combined set of graphs for the simulation. Initially, both the first and second moment is stored, in order to allow calculation of the sampling error in each quantity. These are averaged over the higher level ensembles, to allow estimates of sampling errors. Each resulting graph or average data is each stored in an array of size

`data - all graphics datasets from one sequence member collected in a cell array`
Cell Array, has dimension: `data{graphs}`, made up of a collection of arrays:

1. graph: observable or function making up a single graph with

   Array, has dimension: `(lines, points(1),..,points(d), errorchecks)`.

In the simplest case, there is just one vector component per average. More generally, the number of components is larger than this if there is a requirement to compare different lines in one graph. Note that, unlike the propagating field, the time dimension is fully expanded. This is necessary in order to generate outputs at each of the `points(1)` time slices. Thus, all the space-time dimensions are stored.

When step-size checking is turned on using the *`checks`* flag set to `1`, 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 observables which are stored have three check indices which are all included in the 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.

Because of the error-checking, the last data dimension, errorchecks, is the total number of components in the data array due to error-checking. After ensemble averaging, this index is typically `c = 1, 2, 3`, which 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. As a result, the output `data` ready for graphing with xGRAPH includes step-size error bars and plotted lines for the two estimated upper and lower standard deviations, obtained from the statistical moments.

5.3 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:

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.

xinpreferences()
: is called by *xlattice()* to set the defaults that are not already entered.

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.

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 `in.checks = 0,1`. The integration is executed only once with `in.checks = 0`. With `in.checks = 1`, there is another error-checking integration, 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 *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 either Fourier space or finite differences 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.

5.3.1 Simulation user functions

initial()

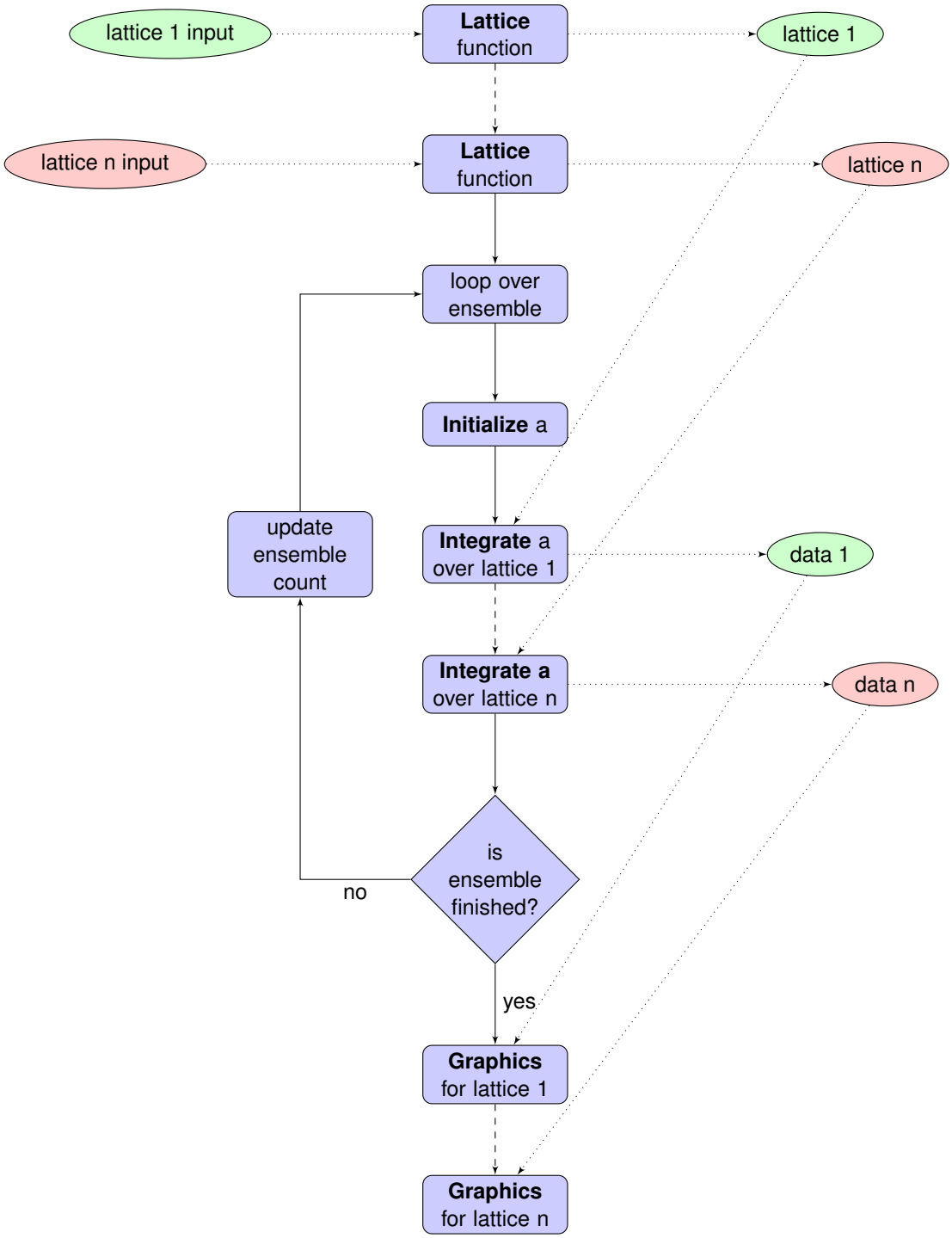

Fig. 5.1: xSPDE flowchart, showing the data, lattice and field processing.

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`. Initialization functions can use coordinates, `r.t`,‘‘r.x‘‘, `r.y`, `r.z`, or for larger dimensions, using numerical lattice labels `r.x{1}`, `r.x{2}`, `r.x{3}`, `r.x{4}`. Numerical labels can be used for any number of dimension if the switch `numberaxis=1`. The default is *xinitial()*, which sets fields to zero.

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 `in.step = xRK4`.

observe()

is a cell array of observation functions whose output is averaged over the ensembles, called from *xpath()*. In general, this returns an array whose first coordinate is the line-number of the n-th graph. The default, *xobserve()*, returns the real amplitudes. The return value is averaged over the local ensemble and stored as data, `d{n}`. Note that the input of *observe()* is the complete field array.

function()

is a cell array of functions used when graphs are needed that are functions of the observed averages. The default value is simply `d{n}`. This is further averaged over higher ensembles to obtain sampling error estimates. Note that the input of *function()* is the complete data cell array, `d`, which includes all the space-time averages for all the observe functions available.

linear()

is the linear response, including transverse derivatives in space. The default, *xlinear()*, sets this to zero. Derivatives are specified using arrays `r.Dx`, `r.Dy`, `r.Dz`, or for larger dimensions, using numerical lattice labels `r.D{2}`, `r.D{3}`, `r.D{4}`, `r.D{5}`.

da()

is called by *step()* to calculate derivatives at every step in the process, including the stochastic terms. Returns a vector with `in.fields(1)` first components.

define()

is called by *step()* to calculate auxiliary fields at every step in the process. Returns a vector with `in.fields(2)` first components.

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

The xSPDE data and arrays that are user accessible are parameters `r`, fields `a`, average observables `data`, and raw trajectories `rawdata`. Apart from the parameters, which are Matlab structures, all fields and data are arrays.

5.4 Data arrays and ensembles

There is a unified index model in all xSPDE arrays. However, in the internal calculations of derivatives and observables, these indices are flattened to give a matrix, as explained below. In all cases, the underlying xSPDE array index ordering is kept exactly the same:

1. field index i
2. time, t index j_1
3. x index j_2

4. y index j_3
5. z index j_4 ..
6. ensemble or error-checking index e or c

The number of space dimensions is arbitrary. To conserve storage, one time - the current one - is stored for propagating fields. The ensemble index can be adjusted to increase or decrease local memory usage. If needed, all data generated can be saved in `rawdata` arrays.

The fields `a` are complex arrays stored discretely on space or momentum grids. Internally, the fields are matrices stored on the flattened xSPDE internal lattice, with just two indices only. Transformations to Fourier space are used both for interaction picture propagation [Caradoc-Davies2000] and for averages over Fourier space.

Two different types of Fourier representations are used. In xsim, Fourier transformations are for propagation, which requires the fastest possible methods, and uses $k = 0$ as the first or lowest index. In xgraph, Fourier transformations are for graphical representations. Hence, the indices are re-ordered to a conventional index ordering, with negative momentum values in the first index position.

The *parameters* are stored in a structure called, simply, `r`. It is available to all user-definable routines. The label `r` is chosen here for no special reason, and can be changed by the user. These structures reside in a static internal cell array that combines both input and lattice parameters, including the interaction picture transformations. The data is generally different for each simulation in a sequence.

Averaged results are called observables in xSPDE. For each sequence, these are stored in either space or Fourier domains, in the array `data`, as determined by the *`transforms`* vector for each observable. This is a vector of switches for each of the space-time coordinates. The `data` arrays obtained in the program as calculations progress are stored in cell arrays, `cdata`, indexed by a sequence index.

If required, `rawdata` 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 *`raw`* option is set to `1`.

All calculated data, including fields, observables and graphics results, is stored in arrays of implicit or explicit rank (2+d), where d is the space-time dimension given in the input. The first index is a field index (i), while the next indices $j \equiv j_1, \ldots j_d \equiv \mathbf{j}$ are for time and space, and the last is a statistics/errors index (e). The space-time dimension d is unlimited.

5.4.1 xSPDE flattened arrays

When the fields, noises or coordinates are integrated by the xSPDE integration functions, they are flattened to a matrix. The first index is the field index, and the combined second index covers all the rest. It is more convenient when calculating derivatives and observables in xSIM, to use these flattened arrays or matrices. They are obtained by combining indices $(\mathbf{j}, e)$ into a flattened second index J. This is faster and more compact notationally. Hence, when used in xSPDE functions, the fields are indexed as $a(i, J)$.

5.4.2 xSPDE array types

There are several different types of arrays used. Note that for the field, noise and coordinate arrays, only one time index is stored, so $j_1 = 1$. The stored ensemble index is for the lowest level statistical ensemble, e_1. These arrays are as follows:

- Field arrays, $a(i, \mathbf{j}, e_1)$ - these have an ensemble index of up to $e_1 = ensembles(1)$, but just a single point in time for efficiency. The fields are flattened to give $a(i, J)$.
- Random and noise arrays, $w(n, \mathbf{j}, e_1)$ - these are like field arrays, except that they contain random numbers for the stochastic equations. Random and noise fields are flattened to give $w(n, J)$, where *n* ranges over the available number of noise variables.

- Coordinate arrays $r.x\{l\}(1,\mathbf{j},e_1)$ - these store the values of coordinates at grid-points, depending on the axis $l = 2,\dots d$, and are part of the main internal data structure, *r*. These only have a single first index. Coordinates are flattened to give $r.x\{l\}(1,J)$. For less than four total dimensions, this notation is replaced by $r.t$,:math:*r.x(1,J)*,:math:*r.y(1,J)*,:math:*r.z(1,J)*. There is a similar array in momentum space, $k\{l\}(1,J)$.
- Raw arrays, $r\{s,c,e_2\}(i,\mathbf{j},e_1)$ - like fields, but with all points stored. Use with care, as they take up large amounts of memory! Here, we use the notation that $\mathbf{j} = j_1, j_2, \dots j_d$ for d space-time dimensions. Note that when output or saved, these have additional cell indices: $s = 1,\dots S$ is the sequence number, $c = 1,2$ for error-checking the time-step, and $h = 1,\dots e_2 * e_3$ for the combined serial and parallel ensemble index. To keep track of all data, an error-check and ensemble index are needed here.
- Data arrays, $d\{g\}(\ell,\mathbf{j},c)$ - these store the averages, or arbitrary functions of them, with an error-checking index $c = 1,2,3$, to store checking data at all time points. Here g is the graph index, $\ell' is the line index. No ensemble index is needed, as these are already ensemble averaged at the first level, so the last index is use$ $math$:mathbf{j} = j_1,j_2,ldots j_d‘ space-time points. If the field is transformed, the **j** flag.
- Graphics data arrays, $gd\{s\}\{g\}(\ell,\mathbf{j},c)$ - these store the data that is actually plotted, and can include further functional transformations if required.

The first index ℓ in a graphics or data array describes different lines on a graph. There can be different first dimensions between fields, noises and output data, as they are specified using different parameters. For only a single output graph, the cell index is not needed.

All outputs have an extra high-level cell index $\{g\}$ called the graph or function index. This corresponds to the index $\{g\}$ of the observe function used to generate averages. One can have several data arrays in a larger cell arrays to make a number of distinct output graphs labelled g, each with multiple averages. Sequences generate separate graphics arrays in sequence, labelled by the first graphics cell index.

More details of ensembles, grids and the internal lattice are given below. Note that the term `lattice` is used to refer to the total internal field storage. This combines the local ensemble and the spatial grid together.

5.4.3 Ensembles

Ensembles are used for averaging over stochastic trajectories. They come in three layers: local, serial and parallel, in order to optimize simulations for memory and for parallel operations. The `in.ensembles(1)` local trajectories are used for array-based parallel ensemble averaging, indexed by e_1. These trajectories are stored in one array, to allow fast on-chip parallel processing.

Distinct stochastic trajectories are also organized at a higher level into a set of `in.ensembles(2)` serial ensembles for statistical purposes, which allows a more precise estimate of sampling error bars. For greater speed, these can be integrated using `in.ensembles(3)` parallel threads. In raw data, these are combined and indexed by the e_2 cell index.

This hierarchical organization allows allows flexibility in allocating memory and optimizing parallel processing. It is usually faster to have larger values of `in.ensembles(1)`, but more memory intensive. Using larger values of `in.ensembles(2)` is slower, but requires less memory. Using larger values of `in.ensembles(3)` is fast, but requires the Matlab parallel toolbox, and uses both threads and memory resources. It is generally not effective to increase `in.ensembles(3)` above the maximum number of available computational cores.

In summary, the stochastic ensembles are defined as follows:

1. Local ensemble: The first or local ensemble contains `ensembles(1)` trajectories stored on the xSPDE internal lattice and processed using matrix operations. These are averaged using vector instructions, and indexed locally with the e_1 index.
2. Serial ensemble: The second or serial ensemble contains `ensembles(2)` of the local ensembles, processed in a sequence to conserve memory.

3. Parallel ensemble: The third or parallel ensemble contains `ensembles(3)` of the serial ensembles processed in parallel using different threads to allow multi-core and multi-CPU parallel operations. The serial and parallel ensembles are logically equivalent, and give identical results. They are indexed by the combined e_2 cell index in raw data.

5.5 Coordinates, integrals and derivatives

5.5.1 Time and space

The default space-time grid for plotted output data is rectangular, with

```
dx(i) = in.ranges(i) / (in.points(i) -1)
```

The time index is `1`, and the space index `i` ranges from `2` to *dimension*. The maximum space-time dimension is unlimited, while `in.ranges(i)` is the time and space duration of the simulation, and `in.points(i)` is the total number of sampled points available in the `i`-th direction. The input `in.boundaries=-1,0,1` changes the space boundary condition, and is given independently for each field, dimension and boundary. The inputs are `-1` for Neumann or specified derivative boundaries (also used for time), `0` for periodic boundaries (the default value) and `1` for Dirichlet or vanishing field boundaries.

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

```
dt = dx(1) / (in.steps * nc)
```

Here, *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.

The xSPDE space and momentum grid can have any dimension, provided there is enough memory. However, default label values are limited to ten, since more than ten total dimensions will require very large time and storage requirements, and is seldom practical unless the grid is extremely coarse.

5.5.2 Space grid

We define the grid cell size dx_j in the j-th dimension in terms of maximum range r_j, the number of points n_j :, and the boundary value r_j, as:

$$dx_j = \frac{r_j}{n_j + b_j}.$$

Each grid starts at a value defined by the vector *origin*. Using the default values, the time grid starts at $t = 0$ and ends at $t = T = r_1$, for $n = 1, \ldots N_j$:

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

Unless there is an offset origin , the j-th coordinate grid starts at $-r_j/2$ and ends at $r_j/2$, so that, for $n = 1, \ldots n_j$:

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

5.5.3 Momentum grid

All fields can be transformed into Fourier space for taking averages in the *observe()* function. This is achieved with the user-defined *transforms* cell array. This is a cell array of vector switches. For any graph and dimension where *transforms* is set to unity, the corresponding Fourier transform is taken.

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

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

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 :

$$dk_j=\frac{2\pi}{dx_jN_j}.$$

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 when graphing the data:

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

However, due to the standard definitions of discrete Fourier transforms, the order used during computation and stored in the data arrays is different, namely:

$$k_j(n)=0..(N_j-1)/2)dk_j,-(N_j-1)/2)dk_j,.-dk_j$$

5.5.4 Averages

There are functions available in xSPDE for grid averages, spatial integrals and derivatives to handle the spatial grid. These can be used to calculate observables for plotting, but are also available for calculating stochastic derivatives as part of the stochastic equation. They operate in parallel over the local ensemble and lattice dimensions. They take a vector or scalar quantity, for example a single field component, and return an average, a space integral, and a spatial derivative respectively. In each case the first argument is the field, the second argument is a vector defining the type of operation, and the last argument is the parameter structure, `r`. If there are only two arguments, the operation vector is replaced by its default value.

Spatial grid averages can be used to obtain stochastic results with reduced sampling errors if the overall grid is homogeneous. An average is carried out using the builtin xSPDE function *xave()* with arguments `(o, [av, ] r)`.

This takes a vector or scalar field or observable, for example `o = [1, n.lattice]`, defined on the xSPDE local lattice, and returns an average over the spatial lattice with the same dimension. The input is a field or observable `o`, and an optional averaging switch `av`. If `av(j) > 0`, an average is taken over dimension `j`. Space dimensions are labelled from `j = 2 ... 4` as elsewhere. If the `av` vector is omitted, the average is taken over all space directions. To average over the local ensemble and all space dimensions, use `xave(o)`. Averages are returned at all lattice locations.

Higher dimensional graphs of grid 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 depends on which dimensions are averaged.

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

```
in.pdimension = 1
```

Note that *xave()* on its own gives identical results to those calculated in the *observe()* functions. Its utility comes when more complex combinations or functions of ensemble averages are required. If the *transforms* switch is set, then momentum space averages are returned.

5.5.5 Integrals

Integrals over the spatial grid allow calculation of conserved or other global quantities. To take an integral over the spatial grid, use the xSPDE function *xint()* with arguments `(o, [dx, ] r)`.

> This function takes a scalar or vector quantity `o`, and returns a trapezoidal 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, ...` 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` to `[0, 0, r.dx(3), r.dx(4)]`, and:

```
in.pdimension = 2
```

If momentum-space integrals are needed, use the *transforms* switch to make sure that the field is Fourier transformed, and input *dk* instead of *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]`.

5.5.6 Derivatives in equations

xSPDE can support either spectral or finite difference methods for derivatives. The default spectral method used is a discrete Fourier transform, but other methods can be added, as the code is inherently extensible. These derivatives are obtained through function calls.

The code to take a spectral derivative, using spatial Fourier transforms, is carried out using the xSPDE *xd()* function with arguments `(o, [D, ] r)`. This can be used both in calculating derivatives for equations, and for averages or observables if they are needed.

This function takes a scalar or vector quantity `o`, and returns a spectral derivative over selected dimensions with a derivative `D`, by Fourier transforming the data. Set `D = r.Dx` for a first order x-derivative, `D = r.Dy` for a first order y-derivative, and similarly `D = r.Dz.*r.Dy` for a cross-derivative in `z` and `y`. Higher derivatives require powers of these, for example *D = r.Dz.^4'*. For higher dimensions use numerical labels, where `D = r.Dx` becomes `D = r.D{2}`, and so on. Time derivatives are ignored at present. Derivatives are returned at all lattice locations.

If the derivative `D` is omitted, a first order x-derivative is returned. Note that *xd()* returns a lattice observable, as required when used in the *observe()* function. If the integral is used in another function, it returns a matrix of dimension `[1, lattice]`.

5.5.7 Finite difference first derivatives

The code to take a first order spatial derivative with finite difference methods is carried out using the xSPDE function *xd1()* with arguments `(o, [dir, ] r)`.

This takes a scalar or vector `o`, and returns a first derivative with an axis direction `dir`. Set `dir = 2` for an x-derivative, `dir = 3` for a y-derivative. Time derivatives are ignored at present. Derivatives are returned at all lattice locations.

If the direction `dir` is omitted, an x-derivative is returned. These derivatives can be used both in calculating propagation, and in calculating observables. The boundary condition is set by the in.boundaries input. It can be made periodic, which is the default, or Neumann with zero derivative, or Dirichlet with zero field.

5.5.8 Finite difference second derivatives

The code to take a second order spatial derivative with finite difference methods is carried out using the xSPDE *xd2()* with arguments `(o, [dir, ] r)` function.

This takes a scalar or vector `o`, and returns the second derivative in axis direction `dir`. Set `dir = 2` for an x-derivative, `dir = 3` for a y-derivative. All other properties are exactly the same as *xd1()*.

5.6 Interaction picture and Fourier transforms

The xSPDE algorithms all allow the use of 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 $\boldsymbol{a}_I(t_n) = \boldsymbol{a}(t_n) \equiv \boldsymbol{a}_n$. It is possible to solve stochastic partial differential equations in xSPDE using explicit derivatives, but this is often less efficient.

A conventional discrete Fourier transform (DFT) using a fast Fourier transform method is employed 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 $\tilde{m} = m - 1$:

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

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

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

We note that the momentum grid spacing is

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

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

$$\boldsymbol{A}(\boldsymbol{k_n}) = \prod_j \left[\sum_{\tilde{m}_j} \exp\left[-i\left(dk_j dx_j\right)\tilde{m}_j\tilde{n}_j\right]\right]$$

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:

$$k_j = (j-1)dk \ \ (j \leq N/2)$$
$$k_j = (j-1-N)dk \ \ (j > N/2)$$

This Fourier transform is multiplied by an appropriate factor to propagate in the interaction picture, than an inverse Fourier transform is applied. While it is for interaction picture transforms, an additional scaling factor is applied to obtain transformed fields in averages.

In other words, in the averages

$$\tilde{a}_n = \frac{dt}{\sqrt{2\pi}} A_{\tilde{n}'}$$

where the indexing change indicates that graphed momenta are stored from negative to positive values. Note also that for frequency spectra a positive sign is used in the frequency exponent, to agree with physics conventions.

5.6.1 Interaction picture derivatives

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

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

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

- $\nabla_x \rightarrow$ `r.Dx`

Here `r.Dx` 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 the equation above. Instead, it is included in the D array. In two dimensions, the code to return a full two-dimensional Laplacian is:

- $\boldsymbol{\nabla}^2 = \nabla_x^2 + \nabla_y^2 \rightarrow$ `r.Dx.^2+r.Dy.^2`

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

5.6.2 Spectra in the time-domain

For calculating a spectrum in the time-domain, the method of inputting a *transforms* switch is used, with `transforms{n}(1) = 1` to turn on Fourier transforms in the time domain for the n-th observable. This requires much more dedicated internal memory.

To conserve memory, one can use more internal *steps* combined with less *points*. In order to ensure that spectral results are independent of memory conservation strategies, xSPDE uses a technique of trapezoidal averaging when calculating frequency spectra.

With this method, all fields are averaged internally using trapezoidal integration in time over any internal steps, to give the average midpoint value. After this, the resulting step-averaged fields are then Fourier transformed.

For example, in the simplest case of just one internal step, with no error-checking, this means that the field used to calculate a spectrum is:

$$\bar{a}_j = (a_{j-1} + a_j)/2,$$

which corresponds to the time in the spectral Fourier transform, of:

$$\bar{t}_j = (t_{j-1} + t_j)/2.$$

For an error-checking calculation with two internal *steps*, there are four successive valuations: a_{j1}, a_{j2}, a_{j3}, a_j, with the last value the one plotted at t_j. In this case, for spectral calculations one averages according to:

$$\bar{a}_j = (a_{j-1} + 2(a_{j1} + a_{j2} + a_{j3}) + a_j)/8.$$

When there are even larger numbers of internal steps, either from error-checking or from using the internal *steps* parameter, one proceeds similarly by carrying out a trapezoidal average over all internal steps.

In addition, one must define the first field $\bar{a}_1$. Due to the cyclic nature of discrete Fourier transforms, this is also logically the last field value. Hence, this is set equal to the corresponding cyclic average of the first and last field value, in order to reduce aliasing errors at high frequencies in the resulting spectrum:

$$\bar{a}_1 = \frac{1}{2}\left(a_N + a_1\right),$$

which corresponds to a time in the spectral Fourier transform of:

$$\bar{t}_1 = t_1 - dt/2 \equiv t_N + dt/2.$$

This aliasing of virtual times, one higher and one lower than any integration time, is a consequence of the discrete Fourier transform method. It also means that the effective total integration time in the Fourier transform definition is $T_{eff} = T + dt = 2\pi/d\omega$, where T is the total integration time, and dt is the time interval between integration points.

5.7 Fields

In the xSIM code, the complex vector field `a` is generally stored as a compressed or flattened matrix with dimensions `[fields, lattice]`. Here `nlattice` is the total number of lattice points including an ensemble dimension, to increase computational efficiency:

```
nlattice =  nspace * ensembles (1)
```

The total number of space points `r.nspace` is given by:

```
nspace = points (2) * ... * points (in.dimension)
```

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

In different subroutines it may be necessary to expand out this array to more easily reference the array structure. However, the internal field structure `a` at a single time-point is as follows

a

** - Array** of dimension: (*fieldsplus*, `nlattice`)

Note: Here, *fieldsplus* = *fields* (1) + *fields* (2) is the total number of field components. Here *fields* (1) are the dynamical fields, while *fields* (2) are defined or auxiliary fields that are sometimes necessary. Also, :attr:`in.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. The time dimension *points* (1) is always compressed to one during calculations. During spectral calculations, and for raw output, the full length of the time lattice, *points* (1), is stored, which increases memory requirements.

5.7.1 Raw output

If required, by using the switch *raw* set to one. xSPDE will then store every trajectory generated. This is raw, unprocessed data, so there is no graph index. The raw data output is stored in an output cell array *raw*. The array is written to disk using the Matlab file-name, on completion, provided a file name is input, and is also available as an xSIM function output.

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

raw

** - Cell Array**, has dimension: `raw{sequence, check, in.ensemble(2)*in.ensemble(3)}`

If thread-level parallel processing is used, these are also stored in the cell array, which is indexed over both the parallel and serial ensemble. Inside each raw cell is at least one complete space-time *a* stored as a complex array, with indices for the field index, the time-space lattice, and the samples.

Each location in the cell array stores one time-space-sample trajectory in xSPDE, which is a real or complex array with (*dimension* + 2) indices, noting that *points* is a vector with *dimension* indices. Here the dynamical fields are expanded to more easily reference the array structure. The full, expanded field structure `a` at a single time-point is as follows

a
: ** - Array** of dimension: (*fieldsplus*, *points*, *ensembles* (1))

Here, *fieldsplus* = *fields* (1) + *fields* (2) is the total number of field components, where *fields* (1) are the dynamical fields, while *fields* (2) are defined or auxiliary fields. Also, :attr:`in.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, and for raw output, the full length of the time lattice, *points* (1), is stored, which increases memory requirements.

The main utility of raw data is for storing data-sets from large simulations for later re-analysis. It is also 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, one might need to plot histograms of distributions from this.

CHAPTER

SIX

XGRAPH

The graphics program *xgraph()* inputs data from *xsim()* simulations, then graphs them in a variety of multidimensional gra

If required, the first argument can be a data file-name. The specified file is then read both for `input` and `data`. The stored input parameters in the file can be replaced by any of the the new `input` parameters that are specified.

6.1 Input and data arrays

To explain xGRAPH in full detail,

- Data inputs are stored in the `data` cell array.
- This describes a *sequence* of simulations, so that `data = {dat1, dat2, ...}`.
- Each structure `data` describes a simulation, with separate graphs for each one.
- The main function is called using `diff = xgraph(data[,input])`.
- The optional second input argument is to allow modification of the graphs.
- The diff output is available when there are comparisons made on the graphed data.

The input data sequence `cdata` is a cell array with a number of individual simulation objects `dat1,...`. Each includes all the parameters that specify the simulation, together with the generated data. If there is only one simulation, just one individual `dat` is needed.

6.2 Customization options

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

Customization options include functions and parameters to allow user definition of:

- graphed functions of the input data
- plot ranges and axes for each dimension
- plot dimensions, ie, two or thee dimensional
- plot types and line types
- comparisons with user-specified functions
- error bars

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

6.3 Graphics functions

To allow options for taking averages, these are carried out in two stages. The first type of average is a local average, taken over any function of the locally stored ensemble of trajectories. These use the :func:observe functions, specified by the user. The default is the real values of each of the fields, stored as a vector. Multiple observe functions can be used, and they are defined as a cell array of functions.

Next, any function can be taken of these local averages, using the :func:function transformations, again specified by the user. The default is the original set of local averages. This is useful if different combinations are needed of the local averages. These second level function outputs are then averaged again over a second level of ensemble averaging, if specified. This is used to obtain estimates of sampling and step-size errors in the final data outputs.

6.3.1 Graphics transforms

All transforms defined in the observables are obtained from a cell array of vectors called *transforms*, which determines if a given coordinate axis is to be transformed prior to a given observable being measured. This can be turned on and off independently for each observable and axis. The coordinate axes are specified in the order of `t,x,y,z,...` This must match the *transforms* attributes used to generate the data, as no additional transform takes place.

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:

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

Accordingly, as explained above, all momentum indices for observable data and axes are re-ordered when graphing, although they are initially stored in the computational order.

6.4 Sequenced observables: `data`

For graphics input, cell data from each simulation in a sequence is packed into successive cells of an overall cell array :data: *data* . This is used to store the total graphics data in a sequence of simulations. All these fields are resident in memory, and can be stored for re-use. They can be re-accessed and replotted, using the *xgraph()* function, if required, with array dimensions:

1. graph: observable or function making up a single graph

   Array, has dimension: `(components, in.points(1), ... in.points(in.dimension), checks)`.

The cell index enumerates first the sequence number and then the graph number. The last 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.

In summary, 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. There are *graphs* real observables, which are determined by the number of functions defined in the *observe()* cell array, unless there are additional functional transformations. The number of *graphs* may be smaller or larger than the number of vector fields. The stored cdata includes all the necessary averages over the ensembles in a complete sequence.

6.4.1 Graphics function

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

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

Comparison results are calculated if available from the user-specified *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.

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. The graphics axes that are used for plotting, and the points plotted, are defined using the optional axes input parameters, where *axes* indicates the axes preferences for n-th graph or set of generated graphical data.

If there are no *axes* inputs, or the inputs are zero - for example, `in.axes{1} = {0,0,0}`, then only the lowest dimensions are plotted, up to 3. If the axes inputs project out a single point in a given dimension, - for example, `axes{1}={0,31,-1,0}`, these axes are suppressed in the plots. This reduces the effective dimension of the data - in this case to two dimensions.

Axis labels can be of three types. There are automatic labels generated of form `t,x,y,z` for up to four space-time dimensions, then labeled axes with indices with more than four dimensions. Thirdly, user defined axis labels are also possible, as well as user defined labels for graphs and for individual quantities plotted.

Inside Matlab, all graphs can be re-edited.

Examples:

- `axes{1}={0}` - For function 1, plot all the time points; higher dimensions get defaults.
- `axes{2}={-1,0}` - For function 2, plot the maximum time (the default), and all x-points. The first or time axis is suppressed.
- `axes{3}={1:4:51,32,64}` - For function 3, plot every 4-th time point at x point 32, y point 64
- `axes{4}={0,1:4:51,0}` - For function 4, plot all time points, every 4-th x point, and all y-points.

Note that -1 indicates a default point, which is the last point on the time axis, and the midpoint on the other axes. This has the effect of suppressing this dimension in any plots.

The pdimension input can also be used to reduce dimensionality, as this sets the maximum effective plotted dimension. For example, `pdimension{1}=1` means that only plots vs time are output for the first function plotted. This is equivalent to setting `axes{1}={0,-1,-1,-1,-1}`. Note that in the following, t,x,y,z are replaced by corresponding higher dimensions if there are axes that are suppressed. Slices can be taken at any desired point, not just the midpoint. Using the standard notation of, for example, `axes{1}={6:3:81}`, can be used to modify the starting, interval, and finishing points for complete control on the plot points.

Results depend on the value of *dimension*, or else the effective graphics dimension if axes are suppressed:

- `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.
- `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.
- `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.
- `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 (1), grey-scale colour plots (2), contour plots (3) and pseudocolor plots (4).

6.4.2 Graphics user functions

gfunction

This is used when a graph is needed that is a function of the data coming from the simulation package, since this data can be analysed at a later time. Error estimates are less accurate when this function is used, due to error-propagation effects that may occur after averaging, unless corrected for explicitly in the graphics function.

xfunctions

This is used when a graph is needed whose axes are a function of the original axes.

compare

This is used when a two-dimensional graph is needed with a comparison line.

6.4.3 Error checks

The final 2D output graphs will have error-bars if *checks* is set to `1`, 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 `in.checks = 0` or increasing *minbar* in final graphs.

If `in.ensembles(2) > 1` or `in.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 `in.ensembles(1)`, which increases the trajectories in a single thread. An alternative is to increase `in.ensembles(2)`. This is slower, but is only limited by the compute time, or else to increase `in.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.

6.4.4 Graphics projections

If there is a spatial grid, 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* is 4 or greater, a central z point `nz = 1 + floor(in.points(4)/2)` is picked. For example, with 35 points, the central point is `nz = 18`.
- This gives a three dimensional space-time lattice, which is treated the same as if *dimension* is 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* is 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* is 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.

CHAPTER

SEVEN

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 five built-in choices of algorithm. All built-in methods are defined in an interaction picture. All can be used with any space dimension, including `in.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 *ipsteps* if the interaction-picture transform length must be changed to a new value.

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

7.1 Notation

The general equation treated here is given in differential form as

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

It is convenient for the purposes of describing interaction picture methods, to introduce an abbreviated notation as:

$$\mathcal{D}\left[\mathbf{a}, t\right] = \boldsymbol{A}\left[\boldsymbol{a}, t\right] + \underline{\mathbf{B}}\left[\boldsymbol{a}, t\right] \cdot \boldsymbol{\zeta}(t).$$

Hence, we can rewrite the differential equation in the form:

$$\frac{\partial \boldsymbol{a}}{\partial t} = \mathcal{D}\left[\mathbf{a}, t\right] + \underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right] \cdot \boldsymbol{a}.$$

Next, we define a linear propagator. This is given formally by:

$$\mathcal{P}\left(\Delta t\right) = \exp\left(\Delta t \underline{\mathbf{L}}\left[\boldsymbol{\nabla}\right]\right)$$

Typically, but not necessarily, this is evaluated in Fourier space, where it should be just a diagonal term in the momentum vector conjugate to the transverse space coordinate. It will then involve a Fourier transform, multiplication by an appropriate function of the momentum, and then an inverse Fourier transform afterwards.

7.2 xSPDE algorithms

The five built-in algorithms provided are:

xEuler()
The first-order Euler method, a simple first-order explicit approach.

xImplicit()
The first-order implicit or backward Euler method, a first-order implicit approach.

xRK2()
A second order Runge-Kutta method.

xMP()
The midpoint method: a semi-implicit, second order algorithm.

xRK4()
A fourth order Runge-Kutta method, which is a popular ODE solver.

For simplicity, the stochastic noise is assumed constant throughout the interval dt. The reader is referred to the literature ([Drummond1991], [Kloeden1995], [Werner1997], [Higham2001]) 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.

Higher order, more complex algorithms for stochastic integration do exist, but they are not included in the current xSPDE distribution. The reason for this is simply that stochastic integration errors are often dominated by the sampling error, which makes the practical advantage of using high-order algorithms less significant in most calculations.

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.

7.3 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 (`in.ipsteps = 1`). Starting from time $t = t_n$, to get the next time point at $t = t_{n+1} = t_n + \Delta t$, one calculates:

$$\Delta \mathbf{a}_n = \Delta t \mathcal{D}\left[\mathbf{a}_n, t_n\right]$$
$$\mathbf{a}_{n+1} = \mathcal{P}\left(\Delta t\right) \cdot \left[\mathbf{a}_n + \Delta \mathbf{a}_n\right]$$

7.4 Implicit

This is an implicit Ito-Euler method using an interaction picture [Drummond1991]. It is more robust, though slower, than the explicit form. 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 implicit Ito form of stochastic equation. It requires one IP transform per step (`in.ipsteps = 1`). Starting

from time $t = t_n$, to get the next time point at $t = t_{n+1} = t_n + \Delta t$, one calculates, using iteration to get the implicit result of the next time-point:

$$\Delta \mathbf{a}_n = \Delta t \mathcal{D}\left[\mathbf{a}_{n+1}, t_n\right]$$
$$\mathbf{a}_{n+1} = \mathcal{P}\left(\Delta t\right) \cdot \left[\mathbf{a}_n + \Delta \mathbf{a}_n\right]$$

7.5 Second order Runge-Kutta

This is a second order Runge-Kutta method using an interaction picture [Caradoc-Davies2000]. 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 (`in.ipsteps = 1`).

To get the next time point, one calculates:

$$\bar{\mathbf{a}} = \mathcal{P}\left(\Delta t\right) \cdot \left[\mathbf{a}_n\right]$$
$$\mathbf{d}^{(1)} = \Delta t \mathcal{P}\left(\Delta t\right) \cdot \mathcal{D}\left[\mathbf{a}_n, t_n\right]$$
$$\mathbf{d}^{(2)} = \Delta t \mathcal{D}\left[\bar{\mathbf{a}} + \mathbf{d}^{(1)}, t_{n+1}\right]$$
$$\mathbf{a}_{n+1} = \bar{\mathbf{a}} + \left(\mathbf{d}^{(1)} + \mathbf{d}^{(2)}\right)/2$$

7.6 Midpoint

This is an implicit midpoint method using an interaction picture. It gives good results for stochastic [Drummond1991] and stochastic partial differential equations [Werner1997]. 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 (`in.ipsteps = 2`).

To get the next time point, one calculates a midpoint derivative iteratively at time to get the next time point at $t = t_{n+1/2} = t_n + \Delta t/2$, to give an estimated midpoint field $\bar{\mathbf{a}}^{(i)}$, usually with three iterations:

$$\bar{\mathbf{a}}^{(0)} = \mathcal{P}\left(\frac{\Delta t}{2}\right) \cdot \left[\mathbf{a}_n\right]$$
$$\bar{\mathbf{a}}^{(i)} = \bar{\mathbf{a}}^{(0)} + \frac{\Delta t}{2}\mathcal{D}\left[\bar{\mathbf{a}}^{(i-1)}, t_{n+1/2}\right]$$
$$\mathbf{a}_{n+1} = \mathcal{P}\left(\frac{\Delta t}{2}\right) \cdot \left[2\bar{\mathbf{a}}^{(i)} - \bar{\mathbf{a}}^{(0)}\right]$$

7.7 Fourth order Runge-Kutta

This is a fourth order Runge-Kutta method using an interaction picture [Caradoc-Davies2000]. 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 (`in.ipsteps = 2`). To get the next time

point, one calculates four derivatives sequentially:

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

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.

7.8 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 `in.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 the total number of ensembles 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

7.9 Extrapolation order and error bars

For checking step-size errors, xSPDE allows the user to specify `checks = 1`, 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 using the more accurate fine step-size results, but with the coarse time lattice in order to calculate the estimated discretization errors. The standard error-bar, with no extrapolation, has a half-size equal to the difference of fine and coarse step graphed results.

Importantly, both fine and coarse time-step results employ identical underlying random noise processes, from the same initial random seed. To compensate for the grid size, the coarse time-step uses a sum of two successive fine noise increments. This has the useful advantage that any differences are only from the effects of the time-step on the integration accuracy. If different noises were used - which is not done - part of the error-bar would be just from sampling errors.

To allow for extrapolation, xSPDE allows user input of an assumed extrapolation order called *order*. If this is done, and *checks* are set to 1 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 $R\left(dt\right)$ with step-size dt have an error of size dt^n, that is:

$$R\left(dt\right) = R_0 + E\left(dt\right) = R_0 + k.dt^n.$$

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

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

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:

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

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

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

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

$$R_0 = \left(1 + \epsilon\right) R_1 - \epsilon R_2.$$

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 `in.order = 0`. This gives fine resolution values and error bars without extrapolation, but is generally less accurate than using extrapolation.

7.10 Sampling errors

Sampling error estimation in xSPDE uses sub-ensemble averaging. Ensembles are specified in three levels. The first, `in.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, `in.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.

CHAPTER

EIGHT

PUBLIC API

8.1 High-level xSPDE functions and objects

The high-level xSPDE functions process the input parameters, producing simulation data and graphs. Function parameters are in the order of `input`, which specifies the simulation, then `data`, which is needed for graphics output.

xspde(*input*)
: This is the combined xSPDE function. It accepts a simulation sequence, `input`. This can be a single structure, `in`, or else a cell array of structures, `{in1,in2,..}`, for sequences. Output graphs are displayed. It returns the output `[error, input, data,rawdata]`, where `error` is the sum of simulation errors in *xsim()*, and difference errors found in the *xgraph()* comparisons. If a filename is specified in the input, it writes an output data file including input and all output data. Raw data is stored on request. It calls the functions *xsim()* and *xgraph()*.

xsim(*input*)
: This is the xSPDE simulation function. Like *xspde()*, it accepts input parameters in `input`. It returns `[maxerror, input, data, rawdata]`, where: `maxerror` is the sum of maximum step-size and maximum sampling errors, `input` is the full input structure or cell array for sequences, including default values, and `data` is a cell array of average observables. If the `in.raw` option is used, data for the actual trajectories is output in `rawdata`. This can be run as a stand-alone function if no graphs are required.

xgraph(*data*[, *input*])
: This is the xSPDE graphics function. It takes computed simulation `data` and `input`. It plots graphs, and returns the maximum difference `diff` from comparisons with user-specified comparison functions. The `data` should have as many cells as `input` cells, for sequences. If `data = 'filename.h5'` or `data= 'filename.mat'`, the specified file is read both for `input` and `data`. Here `.h5` indicates an HDF5 file, and `.mat` indicates a Matlab file. When the `data` input is given as a filename, input parameters in the file are replaced by any of the the new `input` parameters that are specified. Any stored `input` can be overwritten when the graphs are generated. This allows graphs of data to be modified retrospectively.

8.1.1 Open object-oriented architecture

As well as extensibility through sequences, which was described in *Averaging and projects*, in the section *Sequential integration*, the architecture of xSPDE allows functional extensions.

The 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 input parameters and methods to the input structures.

This open object orientation is deliberate. An open extensibility permits arbitrary functions to be specified in the `in` structures. All functions and parameters have default values in xSPDE. It is also possible to include user defined

functions, provided they satisfy the API definitions given below. This is achieved simply by including the relevant function handles and parameters in the input metadata.

Internal parameters and function handles from the `in` structures, together with any computed and default parameters and functions, are stored in the *lattice structure* `r`. These are available to all user-defined functions. Use of pre-existing reserved names is not advisable, and the structure `r.c` is always reserved for user constants, parameters and functions if required.

The xSPDE software architecture is intended to be easily extended, and users are strongly encouraged to develop their own libraries and contribute to the xSPDE function pool. Because this generally requires 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,z,dt,r)
    % a = Mystep(a,z,dt,r) propagates a step my way.
    ...
    a = ...;
end
```

These low-level functions are described in detail in the next sections of this chapter.

8.2 Low-level xSPDE functions

The xSPDE program is function oriented: low-level functions are used to define initial conditions, equations and observables, amongst many other things given in detail below. To make this process simpler, argument passing conventions are used. Common parameters are always passed using a lattices structure variable `r` as the last argument.

Functions of a single lattice have arguments in the following order:

- the field `a` or initial random variable `v`;
- the stochastic noise `z` or other fields;
- non-field arguments;
- the lattice structure `r`.

The first argument, `a`, is a real or complex vector field. This is a matrix whose first dimension is the field index, with a range of 1 to *fieldsplus*, with *fieldsplus* = *fields* (1) + *fields* (2). The second dimension is the lattice index.

The second argument, `z`, if needed, is a real random noise, corresponding to ζ in the mathematical notation. This is a matrix whose first dimension is the noise index. The second dimension is the lattice index.

The last function argument is the *lattice structure*, `r`. This contains data about the integration details and lattice, stored as `r.name`. Important constants in the structure are *t*, the current time, and space coordinates, *x*, *y*, *z*. Other data stored in the structure is explained in later chapters.

Functions of multiple lattice sequences take current arguments first, and the oldest arguments last.

8.2.1 Integration arrays

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

The field dimensions for the flattened arrays passed to xSIM integration functions are:

- `a = [r.fieldsplus, r.nlattice]`
- `rv = [r.randoms(1)+r.randoms(2), r.nlattice]`
- `w = [r.noises(1)+r.noises(2), r.nlattice]`
- `r.Dx, r.x, r.kx = [1, r.nlattice]`

8.2.2 Data arrays

Each observable used to generate graph data is defined by a function in a cell array with length *graphs*. There are two stages of averaging. First, an average over a local ensemble at a single time-point is performed using the *observe()* function. Next, if more sophisticated data is required, an optional *function()* is used to transform data.

The first dimension `lines` is initially determined by the *observe()* function. This can be modifed if required by the data transformation *function()*. It is typically one for a single-line graph, but can be greater. The last dimensions in all data arrays is the vector of time-space dimensions: `points = [points(1), ... , points(dimension)]`.

- `d{n} = [lines,1, points]`.

If the optional *function()* method is used to transform data within xSIM, the entire average data cell array from every *observe()* function is passed after local averaging, to allow all transformations. On output from xSIM to xGRAPH, the data arrays are augmented by the addition of error estimates, addressed using the second index.

8.2.3 Simulation parameters

For each simulation in the `input` sequence, the input parameters 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 *xgpreferences()* 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.

8.3 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. 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.

The standard way to input each parameter value is:

```
in.label = parameter
```

The standard way to input each function is:

```
in.label = @function-name
```

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 given below.

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 `.h5` output 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, or else defined inline.

8.3.1 xSIM parameters

version
Default: `'xSIM3.2'`

This sets the current version number of the simulation program. There is typically no need to input this.

```
in.version = 'current version name'
```

name
Default: `'  '`

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

```
in.name = 'your project name'
```

dimension
Default: `1`

The total space-time dimension is labelled, unsurprisingly,

```
in.dimension = 1...4
```

fields
Default: `[1,0]`

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. The first number is the number of real or complex fields that are initialized by the *initial()* function and integrated using the *da()* derivative. The optional second number is the number of real or complex auxiliary fields specified with the *define()* function.

```
in.fields(1,2) = 0, 1, 2, ...
```

fieldsplus

Default: 1

This is the total of stochastic plus defined fields. This is calculated internally: *fieldsplus* = *fields* (1) + *fields* (2).

```
in.fieldsplus = 0, 1, 2, ...
```

noises

Default: *fields* (1)

This gives the number of stochastic noises generated per lattice point, in coordinate and momentum space respectively. Set to zero (`in.noises = 0`) for no noises. This is the number of *rows* in the noise-vector. Noises can be delta-correlated in x-space or in k-space. The second input is the dimension of noises in k-space. It can be left out if zero. This allows use of finite correlation lengths when needed, by including a frequency filter function that is used to modify the noise in Fourier-space. The Fourier-space random variance is defined by the filter function. This takes the noises in Fourier space, and returns a filtered version, which is inverse Fourier transformed before use. The first noise index, noises(1), indicates how many independent noise fields are generated, while noises(2) indicates how many noises are fourier-transformed, filtered and then inverse fourier transformed to give correlations. These appear as extra noises, so the total is noises(1)+noises(2). The filtered noises have a finite correlation length.

```
in.noises = [in.noises(1), in.noises(2)] >= 0.
```

randoms

Default: *noises*

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 delta-correlated in x-space or in k-space. The second input is the dimension of random fields that are delta-correlated in momentum space. It can be left out if zero. The Fourier-space random variance is modified by the filter function. This takes the randoms in Fourier space, and returns a filtered version, which is inverse Fourier transformed before use. The first noise index, in.randoms(1), indicates how many independent random fields delta-correlated in space are generated, while in.randoms(2) indicates how many additional random fields are fourier-transformed, filtered and then inverse fourier transformed. These are additional random fields, so the total is in.randoms(1)+in.randoms(2). The filtered noises have a finite correlation length.

```
in.randoms = [in.randoms(1), in.randoms(2)] >= 0
```

ranges

Default: `[10, 10, ...]`

Each lattice dimension has a coordinate range, given by:

```
in.ranges = [in.ranges(1), ..., in.ranges(dimension)]
```

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

points

Default: `[51, 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 = [in.points(1), ..., in.points(in.dimension)]
```

The default values are simply given as a rough guide for initial calculations. Large, high dimensional lattices take more time to integrate. Increasing *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.

steps

Default: `1`

Number of time-steps per plotted point. The total number of integration steps in a simulation is therefore `in.steps * (in.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, ...
```

ensembles

Default: `[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 `in.ensembles(2)` or `in.ensembles(3)` are required if sampling error-bars are to be calculated.

```
in.ensembles = [in.ensembles(1), in.ensembles(2), in.ensembles(3)] >= 1
```

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

boundaries

Default: `{[0, 0]}`

Cell array for type of spatial boundary conditions used, set for each dimension independently, and used in the stochastic partial differential equation solutions with finite value derivatives. The cell index is $d = 2, 3, ..$, indicating the dimension. The boundary conditions are defined as a matrix. The first index is the field index i, and the second index the boundary j, with $j = 1$ for the lower and $j = 2$ for the upper boundary. The options are $b = -1, 0, 1$. The default option, or `0`, is periodic. If `-1`, Neumann boundaries are used, with normal derivatives set to zero. If `1`, Dirichlet boundaries are used, with field values set to zero. Note that in the current xSPDE code, setting non-periodic boundaries requires the use of finite difference type derivatives, without the option of an interaction picture derivative. Using Fourier derivatives will make both the boundary conditions periodic, and is not compatible with Neuman or Dirichlet boundaries. Note: boundary values are set by boundfun(a,d,r).

```
in.boundaries{d} = [b1,b2;..] = -1,0,1
```

Indices for setting the boundary conditions are numbered according to the space dimension.

transforms

Default: `{0}`

Cell array that defines the different transform spaces used to calculate observable `n`. This has the structure

```
in.transforms{n} = [t(1), ..., t(4)] >= 0
```

There is one transform vector per observable. The `j`-th index, `t(j)`, indicates a Fourier transform on the `j`-th axis if set to one, starting with the time axis. The default value is zero, indicating no transform. 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 $1/\sqrt{2\pi}$ in each transformed dimension. This gives a Fourier

integral which is symmetrically normalized in ordinary and momentum space. The Fourier transform that is graphed is such that $k = 0$ is the *central* value. The default is that there is no transform used.

scatters

Default: {0}

Cell array that defines the number of scatter trajectories plotted for observable n. This has the structure

```
in.scatters{n} = s >= 0
```

If zero, the mean of the observable is calculated as usual. If nonzero, a set of s observables that correspond to independent stochastic fields are accumulated, with no averaging.

probability

Default: {0}

Cell array that defines the probability plotted for observable n. This has the structure

```
in.probability{n} = o1:o2:o3;
```

If zero, the mean of the observable is calculated as usual. If nonzero, the probability of the observable is calculated and plotted according to the specified vector of axis points. This sets an extra dimension in the data, with $o1$, $o2$, $o3$, being the start, interval and end of the bins used to accumulate probabilities. Bins are centered at o1, o1+o2,o1+2*o2,..o3,each of the same width o2, and the quantity plotted is the average probability density. An extra space dimension is added to store the probability data.

olabels

Default: {'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'
```

c

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

```
in.c = anything
```

8.3.2 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. The output file-name

8.3.3 Advanced input parameters

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

iterations
Default: 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, ...
```

checks
Default: 1

This defines how many times the integration is carried out for error-checking purposes. If *checks* is *0*, there is one integration, but no checking at smaller time-steps. For error checking, set `in.checks = 1`, which repeats the calculation at a shorter time-step — but with identical noise — to obtain the error bars, taking three times longer overall:

```
in.checks = 0, 1
```

order
Default: 0

This is the extrapolation order, which is **only** used if `in.checks = 1`. The program uses the estimated convergence order to extrapolate to zero step-size, with reduced estimated error-bars. If `in.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: 1 for *xEuler()*; 2 for *xRK2()*; 2 for *xMP()*; 4 for *xRK4()*. Allowed values are:

```
in.order >= 0
```

seed
Default: 0

Random noise generation seed, for obtaining reproducible noise sequences. Only needed if `in.noises > 0`

```
in.seed >= 0
```

graphs
Default: number of observables

This gives the number of observables computed. The default is the length of the cell array of observe functions. Normally, this is not initialized, as the default is typically used. Can be used to suppress data averaging.

```
in.graphs >= 0
```

functions
Default: number of functional transformations

This gives the maximum number of graphs computed, which are functions of the observables. The default is the length of the cell array of observe functions. Normally, this is not initialized, as the default is typically used.

```
in.functions >= 0
```

print
Default: `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 structure as a record.

```
in.print >= 0
```

raw
Default: `0`

Flag for storing raw trajectory data. If this flag is turned on, raw trajectories are stored in memory. The raw data is returned in function calls and also written to a file on completion, if a file-name is included.

```
in.raw >= 0
```

origin
Default: `[0, -in.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 `-in.ranges/2` to give results that are symmetric about the origin:

```
in.origin = [origin(1), ..., origin(4)]
```

ipsteps
Default: `1` for *xEuler()* and *xRK2()*, `2` for *xMP()* and *xRK4()*

This specifies the number of interaction picture steps needed in a full propagation time-step. Default values are chosen according to the setting of *step()*. Can be changed for custom integration methods.

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

file
Default: `''`

Matlab or *HDF5* file name for output data. Includes all data and parameter values, including raw trajectories if `in.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`. For a sequence of inputs, the filename should be given in the first structure of the sequence, and the entire sequence is stored.

```
in.file = 'file-name'
```

8.4 xSIM functions

The structure of xsim makes use of many functions, some of which are internal, and some user supplied. This the the main mechanism for extensibility.

8.4.1 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.

initial (*rv*, *r*)
Default: `xinitial()`

Initializes the fields a for the first simulation in a sequence. The initial Gaussian random field variable, `rv`, has unit variance if `dimension` is `1` or else is delta-correlated in space, with variance `1/r.dv` ($\equiv 1/(dx_2...dx_d)$) for d space-time dimensions. If `randoms` is specified in the input, `rv` has a first dimension of `randoms(1) + randoms(2)`. If not specified, the default for `randoms` is `noises`, and the default of `initial()` is `a = 0`.

transfer (*rv*, *r*, *a0*, *r0*)
Default: `xtransfer()`

Initializes the fields a for subsequent calculations in a sequence, together with any required changes in the lattice a. Therefore, it returns $[a, r]$. Otherwise, this function behaves in a similar way to `initial()`. The function includes the previous field `a0` and lattice `r0`. The default set by `xtransfer()` is `[a,r] = [a0,r]`.

da (*a*, *w*, *r*)
Default: `xda()`

Calculates derivatives da of the equation. The noise vector, `w`, has variance $1/(dx_1..dx_d)$, for dimension $d \leq 4$, and a first dimension whose default value is `fields` if `noises` are not given. Otherwise, it has a first dimension of `in.noises(1) + in.noises(2)`. The second type of input noise allows for spatially correlated and filtered noise specified in momentum space.

define (*a*, *w*, *r*)

Default: `xdefine()`

Calculates auxiliary field values during propagation.

linear (*r*)
Default: `xlinear()`

A user-definable function which returns the linear coefficients L in Fourier space. This is a function of the differential operator `Dx`, `Dy`, `Dz`, which correspond to $\partial/\partial x$, $\partial/\partial y$, $\partial/\partial z$ respectively. Each component has an array dimension the same as the coordinate lattice. If axes are numbered, use `D{1}`, `D{2}`, `D{3}` etc.

observe (*a*, *r*)
Default: cell array of `xobserve()`

Cell array of function handles that take the current field and returns a real observable `o` with dimension of `[1, 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 `graphs`. The fields `a` passed in the input are transformed according to the `functions` metadata.

function (*data*, *in*)
This is a user-definable cell array of data function handles. Use when simulation data is needed that is a function of the `observe()` local averages over `ensemble(1)`, typically involving combinations of several observed averages. The default value sequentially generates all the averages that are in the simulated data. The input to the `n`-th function is the whole cell array of averages, and the output is a data array for the `n`-th graph. This function is used at simulation time. It generates error-bars and sampling errors in the graphed results.

rfilter (*w*, *r*)
Default: `xrfilter()`

Returns the momentum-space filter function for the input random terms. Each component has an array dimension the same as the input random fields in momentum space, that is, the return dimension is `[r.randoms(2), r.nlattice]`.

..function:: nfilter (w,r)

Default: `xnfilter()`

Returns the momentum-space filter function for the propagation noise terms. Each component has an array dimension the same as the random noises in momentum space, that is, the return dimension is `[r.noises(2), r.nlattice]`.

8.4.2 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:

xave (*o*[, *av*], *r*)

This function takes a vector or scalar field or observable, for example `o = [1, n.lattice]`, defined on the xSPDE local lattice, and returns an average over the spatial lattice with the same dimension. The input is a field or observable `o`, and an optional averaging switch `av`. If `av(j) > 0`, an average is taken over dimension `j`. Space dimensions are labelled from `j = 2 ... `` as elsewhere. If the ``av` vector is omitted, the average is taken over all space directions. To average over the local ensemble and all space dimensions, use `xave(o)`. Averages are returned at all lattice locations.

xint (*o*[, *dx*], *r*)

This function takes a scalar or vector quantity `o`, and returns a space integral over selected dimensions with vector measure `dx`. If `dx(j) > 0` an integral is taken over dimension `j`. Space dimensions are labelled from `j = 2, ...` as elsewhere. Time integrals are ignored at present. To integrate over an entire lattice, set `dx = r.dx`, otherwise set `dx(j) = r.dx(j)` for selected dimensions `j`. If the input array is fourier transformed, by using the `transforms` attribute in the `observe` function, then one must set `dx(j) = r.dk(j)` for transformed dimensions `j`, to get correctly normalised results. If the `dx` vector is omitted, the integral is taken over all space directions, assuming no Fourier transforms. Integrals are returned at all lattice locations to give a fixed array size for observables.

xbin (*o*[, *dx*], *r*)

This function takes a scalar or vector quantity `o`, and returns a binned observable on a space axis defined by the vector measure `dx`. The purpose is to allow binning of probabilities in cases when the observable is a mean position, and can be simply plotted on an axis. If `j` is the first index with `dx(j) > 0` the binning is taken over dimension `j`. The results returned are the probability of `o` in the bin, normalized by $1/dx(j)$. Space dimensions are labelled from `j = 2, ...` as elsewhere. If the input array is fourier transformed, by using the `transforms` attribute in the `observe` function, then one must set `dx(j) = r.dk(j)` for transformed dimensions `j`, to get correctly binned results. If the `dx` vector is omitted, or a scalar `dx` is used, the binning is taken over the first space direction.

xnft (*o*[, *trans*], *r*)

This function takes a scalar or vector quantity `o`, and returns a normalized Fourier transform over selected dimensions with Fourier components in the standard order used for graphs. If `trans(j) > 0` a transform is taken over dimension `j`. Space dimensions are labelled from `j = 2, ...` as elsewhere. Time transforms are ignored at present. If the `trans` vector is omitted, the integral is taken over all space directions. Transforms are returned at all lattice locations to give a fixed array size for observables.

xd (*o*[, *D*], *r*)

This function takes a scalar or vector quantity `o`, and returns a spectral derivative over selected dimensions with a derivative `D`, by Fourier transforming the data. Set `D = r.Dx` for a first order x-derivative, `D = r.Dy` for a first order y-derivative, and similarly `D = r.Dz.*r.Dy` for a cross-derivative in `z` and `y`. Higher derivatives require powers of these, for example *D = r.Dz.^4'*. For higher dimensions use numerical labels, where `D = r.Dx` becomes `D = r.D{2}`, and so on. If the derivative `D` is omitted, a first order x-derivative is returned.

xd1 (*o*[, *dir*, *values*], *r*)

This takes a scalar or vector `o`, and returns a first derivative with an axis direction `dir` using finite differences. Set `dir = 2` for an x-derivative, `dir = 3` for a y-derivative, and so on. Time derivatives are ignored at

present. Derivatives are returned at all lattice locations. The boundary condition is set by the in.boundaries(j,dir) input, where `j = 1` indicates the lower, and `j = 2` the upper limit in each direction `dir ``. It can be made periodic (``in.boundaries = 0`), which is the default, or Neumann with fixed derivative `in.boundaries = -1`, or Dirichlet with specified field `in.boundaries = 1`. The default boundary values are zero, if `values` is omitted. If required, they are specified in the calling argument as values(i,j=1,2), where `i` is the component index, `j` is the boundary index.

xd2 (*o*[, *dir*, *values*], *r*)
This takes a scalar or vector `o`, and returns the second derivative in axis direction `dir`. Set `dir = 2` for an x-derivative, `dir = 3` for a y-derivative. All other properties are exactly the same as *xd1()*.

grid(*r*)
Default: xgrid()

Initializes the grid of coordinates in space.

noisegen(*r*)
Default: xnoisegen()

Generates arrays of noise terms `xi` for each point in time.

randomgen(*r*)
Default: xrandomgen()

Generates a set of initial random fields `v` to initialize the fields simulated.

step(*a*, *w*, *r*)
Default: xRK4()

Specifies the stochastic integration routine for the field `a`, noise `w`, together with the interaction-picture propagator *propagator* which is part of the lattice structure. It returns the new field `a`. It uses the current step in time `r.dtr` and current time `r.t`. This function 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, *xRK4()*, is a fourth-order Runge-Kutta. Another very useful alternative, *xMP()*, is a midpoint integrator.

prop(*a*, *r*)
Default: xprop()

Returns the fields propagated for one step in the interaction picture, depending on the initial field `a`, and the propagator array *propagator*. Note that the time-step used in *propagator* depends on the input time-step, the error-checking and the algorithm.

propfactor(*nc*, *r*)
Default: xpropfactor()

Returns the transfer array *propagator*, used by the *prop* function. The time propagated is a fraction of the current integration time-step, *dt*. It is equal to `1 / ipsteps` of the integration time-step.

boundfun(*a*, *d*, *r*)
Default: xboundfun()

For non-vanishing, specified boundary conditions, the boundary function `boundfun(a,d,r)` is called. This returns the boundary values used for the fields or derivatives in dimension d>1 as an array b(i,J,j,K)).

Here i is the field index, J is a flattened dimension equal to the product of field array dimensions less than d, and K is a flattened dimension equal to the product of field array dimensions greater than d, including the ensemble dimension. Crucially, j is the index of the dimension d whose boundary values are specified, so only two values are needed: j=1,2 for the upper and lower boundary values, which are either field values or derivatives.

Boundary values may be constant or a function of both the fields (a) and internal variables like the current time (r.t). The boundary values can have predefined or stochastic initial values which need to be calculated only once. In such

cases the boundary values must first be initialized, so the routine boundfun(a,d,r) is called with time t<origin(1), and the field `a` equal to the random field, delta-correlated in space, used to initialize field evolution. The program uses these results to store values for the boundaries in an internal array, boundval{d}, for later use if needed.

The default initial boundary value is zero, set by the default boundary function xboundfun(a,d,r). The boundary values returned by xboundfun(a,d,r), are set equal to the initial boundary values stored in r.boundval{d}.

8.5 xGRAPH parameters

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 *xgpreferences()* function, in all cases except for the comparison function *compare()*. The defaults can be changed by editing the *xgpreferences()* function.

In the following descriptions, *graphs* 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, that is, if the dimension variable is set to one.

For uniformity, the graphics parameters that reference an individual data object 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. The graph number used to index these cell arrays refers to the data object, and there can be multiple plots obtained, depending on the graphics input.

Together with default values, they are:

gversion
Default: `'xGRAPH3.2'`

This sets the current version number of the graphics program. There is typically no need to input this.

```
in.gversion = 'current version name'
```

graphs
Default: `in.functions`

If specified, this sets the maximum number of graphed datasets. Can be used to suppress unwanted graphs from an xSPDE graphics script. If omitted, all the data output from the in.functions data processing functions are plotted.

```
in.graphs = 1,..
```

olabels
Default: `{'a', ...}`

Cell array of labels for the graph axis observables and 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'

.. attribute:: parametric
```

Default: [0,0]

Cell array that defines parametric plots, if required, for each graph number. The first number is the graph number of the observable plotted on the horizontal axis for the parametric plot. The second number is the axis number where the parametric value is substituted, which can be time (axis 1) or x-coordinate (axis 2).

```
in.parametric{n} = [p1,p2] >= 0
```

If both are zero, the plot against an independent space-time coordinate is calculated as usual. If nonzero, a parametric plot is made, for two-dimensional plots only. In both cases the vertical axis is used to plot the graph variable. The horizontal axis is used for either the independent variable or the parametric variable. In this version, only vertical error-bars are available. Can be usefully combined with in.scatters{n} to plot individual trajectories, but the number of scatters should be the same in each of the two graphs that are parametrically plotted against each other.

axes

Default: {{0,0,0,..}}

Gives the axis and points plotted p for each plotted function. As special cases, p = 0, is the default value that gives the entire axis, while p = -1 generates one point on the axis, namely the last point for the time axis and the midpoint for the space axes. Other values are vector range indicators, for example p = 5 plots the fifth point, while p = 1:4:41 plots every fourth point. For each graph type n the axes can be individually specified. If more than three axes are specified, only the first three are used. The others are set to default values.

```
in.axes{n} = {p1,p2,p3,..pd}
```

font

Default: {18, ...}

This sets the default font sizes for the graph labels, indexed by graph. This can be changed per graph.

```
in.font{n} > 0
```

esample

Default: {1, 1 ...}

This sets the type and size of sampling errors that are plotted. If esample = 0, no sampling error lines are plotted, just the mean. If esample = -n, $\pm n\sigma$ sampling errors are included in the errorbars. If esample = n, upper and lower $\pm n\sigma$ sampling error lines are plotted. In all cases, the magnitude of esample sets the number of standard deviations used.

in.esample{n} = e

minbar

Default: {0.01, ...}

This is the minimum relative error-bar that is plotted. Set to a large value to suppress unwanted error-bars, although its best not to ignore the error-bar information! This can be changed per graph.

```
in.minbar{n} >= 0

.. attribute:: esample
```

Default: {1, ...}

This is the flag for plotting sampling error. Set to zero to suppress unwanted sampling error lines and just plot means, although its best not to ignore this information! This can be changed per graph.

```
in.esample{n} >= 0
```

images

Default: {0, 0, 0, ...}

This is the number of 3D, transverse o-x-y movie images plotted as discrete time slices. Only valid if *dimension* is greater than 2. Note that, if present, the coordinates not plotted are set to their central value, for example z = 0, when plotting the transverse images. This input should have a value from in.images(n) = 0 up to a maximum value of the number of plotted time-points. It has a vector length equal to *graphs*:

```
in.images{n} = 0 ... in.points(1)
```

imagetype

Default: {1, 1, ...}

This is the *type* of transverse o-x-y movie images plotted. If an element is 1, a perspective surface plot is output, for 2, a gray plot with colours is output, for 3 a contour plot with 10 equally spaced contours is generated, and for 4 a pseudocolor map is generated.This has a vector length equal to *graphs*.

```
in.imagetype{n} = 1, 2, 3, 4
```

transverse

Default: {0, 0, ...}

This is the number of 2D, transverse o-x images plotted as discrete time slices. Only valid if *dimension* is greater than 2. Note that, if present, the y,z-coordinates are set to their central values, when plotting the transverse images. Each element should be from 0 up to a maximum value of the number of plotted time-points. It has a vector length equal to *graphs*:

```
in.transverse{n}=0 ... in.points(1)
```

headers

Default: {'head1', 'head2', ...}

This is a string variable giving the graph headers for each type of function plotted. The default value is an empty string '', which gives the overall simulation heading. Use a space ' ' to suppress graphics headers entirely. It is useful to include simulation headers - which is the default - to identify graphs in preliminary stages, while they may not be needed in a final result.

```
in.headers{n} = 'my_graph_header'
```

pdimension

Default: {3, 3, ...}

This is the maximum space-time grid dimension for each plotted quantity. The purpose is eliminate unwanted graphs. For example, it may be 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. All graphs are suppressed if it is set to zero. Any three dimensions can be chosen using the axes command.

```
in.pdimension{n} \ge 0
```

xlabels

Default: {'t', 'x', 'y', 'z'} or {'x_1', 'x_2', 'x_3', 'x_4'}

Labels for the graph axis independent variable labels, vector length of *dimension*. The numerical labeling default is used when the `in.numberaxis` option is set. *Note, these are typeset in Latex mathematics mode!*

```
in.xlabels = {in.xlabels(1), ..., in.xlabels(in.dimension)}
```

klabels

Default: `{'\\omega', 'k\_x', 'k\_y', 'k\_z'}` or `{'k\_1', 'k\_2', 'k\_3', 'k\_4'}`

Labels for the graph axis Fourier transform labels, vector length of *dimension*. The numerical labeling default is used when the `in.numberaxis` option is set. *Note, these are typeset in Latex mathematics mode!*

```
in.klabels = {in.klabels(1), ..., in.klabels(in.dimension)}
```

glabels

Default: `{{'t', 'x', 'y', 'z'}}` or `{{'\omega', 'k_x', 'k_y', 'k_z'}}`

Graph-dependent labels for the independent variable labels, nested cell array with first dimension *graphs*, second dimension *dimension*. This is useful if the axis labels

change from graph to graph, for example, if the coordinates have a functional transform.

```
in.glabels{n} = {in.xlabels(1), ..., in.xlabels(in.dimension)}
```

lines

Default: '' {{'-k','–k',':k','-.k','-ok','–ok',':ok','-.ok','-+k','–+k'}}''

Line types for each line in every two-dimensional graph plotted. If a given line on a two-dimensional line is to be removed completely, set the relevant line-style to zero. For example, to remove the first line from graph 3, set in.lines{3} ={0}. This is useful when generating and changing graphics output from a saved data file.

```
in.lines{n} = {linetype{1}, ..., linetype{nl}}
```

8.6 xGRAPH functions

gfunction (*data*, *in*)

This is a cell array of graphics function handles. Use when a graph is needed that is a functional transformation of the observed averages. The default value generates all the averages that are in the simulated data. The input is the data cell array of averages, and the output is the data array that is plotted. Note that in general the cell index is used to describe a given graph, while the first vector index in the graphed data indexes a line in the graph. For multidimensional data, the graphics program automatically generates several different projections of a given graph to allow a complete picture.

xfunctions (*x_nd*, *in*)

This is a nested cell array of graphics axis transformations. Use when a graph is needed with an axis that is a function of the original axes. The input of the function is the original axis coordinates, and the output is the new coordinate set. The default value generates the input axes. Called as in.xfunctions{n}{nd}(x_nd,in) for the n-th graph and axis nd, where x_nd is a vector of axis coordinate points for that axis dimension.

compare (*t*, *in*)

This is a cell array of comparative functions. Each takes the time or frequency vector - or whichever is the first dimension plotted - and returns comparison results for a graphed observable, as a function versus time or frequency (etc). Comparison results are graphed with a dashed line, for the two-dimensional graphs versus the first plotted dimension. There is no default function handle.

8.7 Parameter structure

Internally, xSPDE parameters and function handles are 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 addition, it includes the table of computed parameters given below.

User application constants and parameters should not be reserved names. No reserved name uses capitals (except `D`), special symbols, or starts with `c`. Therefore, all names starting with `in.c`, special symbols or capitals - (except `D` for derivatives) - will be available to the user name-space in future versions of xSPDE.

A parameter structure contains information about the space-time grid and is passed to various functions, for instance `da()` or `step()`. The corresponding parameter is accessed in the structure *r*, for example, *r.x*.

t
: Current value of time, t.

x

y

z
: Coordinate grids of x, y, z.

r{n}
: Higher dimensions are labeled numerically as r_1,.. r_6, and so on. This numerical axis convention can be set even for lower dimensions if `in.numberaxis` is set to 1.

kx

ky

kz
: Grids in momentum space: k_x, k_y, k_z.

k{n}
: Higher dimensions are labeled numerically as k_5, k_6, and so on.

dt
: Output time-step between stored points for data averages.

dtr
: Current reduced time-step used for integration.

dx
: Steps in coordinate space: $[t, x, y, z, x_5, ..]$.

dk
: Steps in momentum space: $[\omega, k_x, k_y, k_z, k_5, ..]$.

propagator
: Contains the propagator array for the interaction picture.

v
: Spatial lattice volume.

kv
: Momentum lattice volume.

dv
: Spatial cell volume.

dkv
: Momentum cell volume.

xc
Space-time coordinate axes (vector cells).

kc
Computational Fourier transform axes in $[\omega, k_x, k_y, k_z, k_5, ..]$ (vector cells).

kg
Graphics Fourier transform axes in $[\omega, k_x, k_y, k_z, k_5, ..]$ (vector cells).

kranges
Range in $[\omega, k_x, k_y, k_z, k_5, ..]$ (vector).

`s.`**dx**
Initial stochastic normalization.

`s.`**dxt**
Propagating stochastic normalization.

`s.`**dk**
Initial k stochastic normalization.

`s.`**dkt**
Propagating k stochastic normalization.

nspace
Number of spatial lattice points: `in.points(2) * .. * in.points(in.dimension)`.

nlattice
Total lattice: `in.ensembles(1) * nspace`.

ncopies
Total copies of stochastic integrations: `in.ensembles(2) * in.ensembles(3)`.

`d.`**int**
Dimensions for lattice integration (vector).

`d.`**a**
Dimensions for a field (flattened, vector).

`d.`**r**
Dimensions for coordinates (flattened, vector).

`d.`**ft**
Dimensions for field transforms (vector).

`d.`**k**
Dimensions for noise transforms (vector).

`d.`**obs**
Dimensions for observations (vector).

`d.`**data**
Dimensions for average data (flattened, vector).

`d.`**raw**
Dimensions for raw data (flattened, vector).

8.8 Default functions

These functions are used as defaults for simulations and can be overridden by the user.

xinitial (~, *r*)
Returns a field array filled with zeros.

xtransfer (~, ~, *a*, ~)
Returns the field `a` unchanged.

xda (~, ~, *r*)
Returns a derivative array filled with zeros.

xdefine (~, ~, *r*)
Returns a define array filled with zeros.

xlinear (~, *r*)
Returns a linear response array filled with zeros.

xobserve (*a*, ~)
Returns the real part of `a(1,:)`.

xrfilter (*v*, *r*)
Returns the unfiltered random field v.

xnfilter (*w*, *r*)
Returns the unfiltered noise w.

xgrid (*r*)
Sets grid points in lattice from coordinate vectors. Returns the `r` structure with added grid points.

xnoisegen (*r*)
Generates random noise matrix z.

xrandomgen (*r*)
Generates default initial random field matrix v.

xpropfactor (*nc*, *r*)
Returns default interaction picture propagation factor. `nc` is a check index, `r` is a lattice structure.

xboundfun (*a*, *d*, *r*)
Returns default initial boundary value of zero. The later time boundary values returned by xboundfun(a,d,r), are set equal to the initial boundary values stored in r.boundval{d}, whose default is initialized to zero by the initial boundary value.

8.9 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 the functions that need to be averaged using the user function *observe()*.
- Can you have functions of the current time and space coordinate?
  - Yes: xSPDE functions support this using the structure `r`, as *t*, *x*, *y*, *z*, or :attr:`x{1}`, and so on, for more than four space-time dimensions.
- Can you have several independent stochastic variables?
  - Yes, input this using `in.fields(1) > 1`.
- I need to have auxiliary fields are defined as functions of other fields.
  - No problem, input this specification using `in.fields(2) > 0`, and define the extra fields with the *define()* function.

- Are higher dimensional differential equations possible?
  - Yes, this requires setting `in.dimension > 1`. This is essentially unlimited in xSPDE except for memory requirements.
- Can you have spatial partial derivatives?
  - Yes, provided they are linear in the fields; these are obtainable using the function *`linear()`*. If they are more general, use the derivative functions *`xd()`* or or you want special, nonperiodic boundary conditions, then use the finite difference methods, *`xd1()`* and *`xd2()`* .
- Can you delete the graph heading?
  - Yes, this is turned off if you set *`headers`* to `0`.
- Why are there two nearly parallel lines in the graphs sometimes?
  - These are one standard deviation sampling error limits, generated when `in.ensembles(2,3) > 1`.
- Why is there just one line in some graphs, with no sampling errors indicated?
  - You need `in.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.

CHAPTER

NINE

REFERENCES

CHAPTER

TEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Gardiner2004] C. W. Gardiner, *Handbook of Stochastic Methods: for Physics, Chemistry and the Natural Sciences* (Springer 2004).

[Colecutt2001] G. R. Collecutt, P. D. Drummond, *XMDS: eXtensible multi-dimensional simulator.* Comput. Phys. Commun. **142**, 219-223 (2001).

[Dennis2013] Graham R. Dennis, Joseph J. Hope, Mattias T. Johnsson, *XMDS2: Fast, scalable simulation of coupled stochastic partial differential equations,* Comput. Phys. Commun. **184**, 201208 (2013).

[Drummond1991] P. D. Drummond and I. K. Mortimer, *Computer simulations of multiplicative stochastic differential equations.* J. Comp. Phys. **93**, 144-170 (1991).

[Kloeden1995] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations* (Springer, 1995).

[Werner1997] M. J. Werner and P. D. Drummond, *Robust algorithms for solving stochastic partial differential equations.* J. Comp. Phys. **132**, 312-326 (1997).

[Caradoc-Davies2000] B. M. Caradoc-Davies, *Vortex dynamics in Bose-Einstein condensates* (Doctoral dissertation, PhD thesis, University of Otago (NZ), 2000).

[Higham2001] Desmond J. Higham, *An Algorithmic Introduction to Numerical Simulation of Stochastic Differential Equations*, SIAM Review **43**, 525546 (2001).

INDEX