Equation-Free function toolbox for Matlab/Octave:

Summary User Manual

A. J. Roberts* John Maclean† J. E. Bunder‡

May 2, 2023

* School of Mathematical Sciences, University of Adelaide, South Australia. https://profajroberts.github.io, http://orcid.org/0000-0001-8930-1552
† School of Mathematical Sciences, University of Adelaide, South Australia. http://www.adelaide.edu.au/directory/john.maclean
‡ School of Mathematical Sciences, University of Adelaide, South Australia. mailto:judith.bunder@adelaide.edu.au, http://orcid.org/0000-0001-5355-2288

Abstract

This 'equation-free toolbox' empowers the computer-assisted analysis of complex, multiscale systems. Its aim is to enable you to use microscopic simulators to perform system level tasks and analysis, because microscale simulations are often the best available description of a system. The methodology bypasses the derivation of macroscopic evolution equations by computing only short bursts of the microscale simulator (Kevrekidis & Samaey 2009, Kevrekidis et al. 2004, 2003, e.g.), and often only computing on small patches of the spatial domain (Roberts et al. 2014, e.g.). This suite of functions empowers users to start implementing such methods in their own applications. Download via https://github.com/uoa1184615/EquationFreeGit

Contents

1 Introduction

Users Download via https://github.com/uoa1184615/EquationFreeGit. Place the folder of this toolbox in a path searched by MATLAB/Octave. Then read the section(s) that documents the function of interest.

Quick start Maybe start by adapting one of the included examples. Many of the main functions include, at their beginning, example code of their use—code which is executed when the function is invoked without any arguments.

- To projectively integrate over time a multiscale, slow-fast, system of ODEs you could use `PIRK2()`, or `PIRK4()` for higher-order accuracy: adapt the Michaelis–Menten example at the beginning of `PIRK2.m` (Section 2.2.2).
- You may use forward bursts of simulation in order to simulate the slow dynamics backward in time, as in `egPIMM.m` (Section 2.3).
- To only resolve the slow dynamics in the projective integration, use lifting and restriction functions by adapting the singular perturbation ODE example at the beginning of `PIG.m` (Section 2.4.2).

Space-time systems Consider an evolving system over a large spatial domain when all you have is a microscale code. To efficiently simulate over the large domain, one can simulate in just small patches of the domain, appropriately coupled.

- In 1D space adapt the code at the beginning of `configPatches1.m` for Burgers' PDE (Section 3.1.1).
- In 2D space adapt the code at the beginning of `configPatches2.m` for nonlinear diffusion (Section 4.1.1).
- In 3D space adapt the code at the beginning of `configPatches3.m` for wave propagation through a heterogeneous medium (Section 5.1.1), or the patches of the 3D heterogeneous diffusion of `homoDiffEdgy3.m` (Section 5.4).
- Other provided examples include cases of macroscale *computational homogenisation* of microscale heterogeneity.
- In MATLAB, the axis labelling works best if one executes `set(groot,'defaultTextInterpreter','latex')`

Verification Most of these schemes have proven 'accuracy' when compared to the underlying specified microscale system. In the spatial patch schemes, we measure 'accuracy' by the order of consistency between macroscale dynamics and the specified microscale.

- Roberts & Kevrekidis (2007) and Roberts et al. (2014) proved reasonably general high-order consistency for the 1D and 2D patch schemes, respectively.
- In wave-like systems, Cao & Roberts (2016) established high-order consistency for the 1D staggered patch scheme.
- A heterogeneous microscale is more difficult, but Bunder et al. (2017) showed good accuracy in a variety of circumstances, for appropriately chosen parameters. Further, Bunder et al. (2020) developed a new 'edgy' inter-patch interpolation that is proven to be good for simulating the macroscale homogenised dynamics of microscale heterogeneous systems—now coded in the toolbox.

Blackbox scenarios Suppose that you have a *detailed and trustworthy* computational simulation of some problem of interest. Let's say the simulation is coded in terms of detailed (microscale) variable values $\vec{u}(t)$, in $\mathbb{R}^p$ for some number p of field variables, and evolving in time t. The details $\vec{u}$ could represent particles, agents, or states of a system. When the computation is too time consuming to simulate all the times of interest, then Projective Integration may be able to predict long-time dynamics, both forward and backward in time. In this case, provide your detailed computational simulation as a 'black box' to the Projective Integration functions of Chapter 2.

In many scenarios, the problem of interest involves space or a 'spatial' lattice. Let's say that indices i correspond to 'spatial' coordinates $\vec{x}_i(t)$, which are often fixed: in lattice problems the positions $\vec{x}_i$ would be fixed in time (unless employing a moving mesh on the microscale); however, in particle problems the positions would evolve. And suppose your detailed and trustworthy simulation is coded also in terms of micro-field variable values $\vec{u}_i(t) \in \mathbb{R}^p$ at time t. Often the detailed computational simulation is too expensive over all the desired spatial domain $\vec{x} \in \mathbb{X} \subset \mathbb{R}^d$. In this case, the toolbox functions of Chapter 3 empower you to simulate on only small, well-separated, patches of space by appropriately coupling between patches your simulation code, as a 'black box', executing on each small patch. The computational savings may be enormous, especially if combined with projective integration.

Chapter 6 provides small examples of how to parallelise the patch computations over multiple processors. But such parallelisation may be only useful for scenarios where the microscale code has many millions of operations per time-step.

Contributors The aim of this project is to collectively develop a MATLAB/Octave toolbox of equation-free algorithms. Initially the algorithms are basic, and the plan is to subsequently develop more and more capability.

MATLAB appears a good choice for a first version since it is widespread, efficient, supports various parallel modes, and development costs are reasonably low. Further it is built on BLAS and LAPACK so the cache and superscalar CPU are potentially well utilised. We aim to develop functions that work for MATLAB/Octave.

2 Projective integration of deterministic ODEs

Chapter contents

2.1 Introduction

This section provides some good projective integration functions (Gear & Kevrekidis 2003*b*,*c*, Givon et al. 2006, Marschler et al. 2014, Maclean & Gottwald 2015, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive multiscale dynamic simulations/integrations to efficiently compute over very long time scales.

Quick start Section 2.2.2 shows the most basic use of a projective integration function. Section 2.3 shows how to code more variations of the introductory example of a long time simulation of the Michaelis–Menton multiscale system of differential equations. Then see Figures 2.1 and 2.2

Scenario When you are interested in a complex system with many interacting parts or agents, you usually are primarily interested in the self-organised emergent macroscale characteristics. Projective integration empowers us to efficiently simulate such long-time emergent dynamics. We suppose you have coded some accurate, fine-scale, microscale simulation of the complex system, and call such code a microsolver.

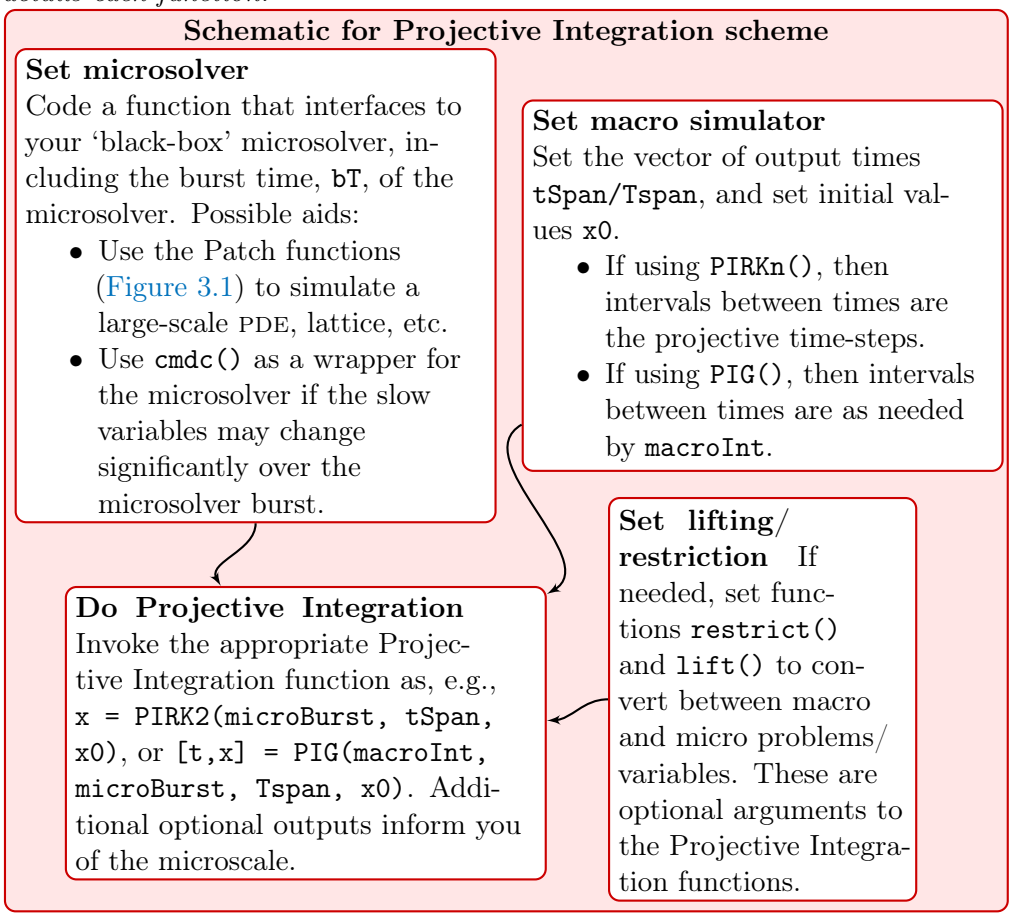
The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

Petersik (2019–) is also developing, in python, some projective integration functions.

Main functions

- Projective Integration by second or fourth-order Runge–Kutta is implemented by `PIRK2()` or `PIRK4()` respectively. These schemes are suitable for precise simulation of the slow dynamics, provided the time period spanned by an application of the microsolver is not too large.
- Projective Integration with a General method, `PIG()`. This function enables a Projective Integration implementation of any integration method over macroscale time-steps. It does not matter whether the method is a standard MATLAB/Octave algorithm, or one supplied by the user. `PIG()` should only be used directly in very stiff systems, less stiff systems additionally require `cdmc()`.
- *Constraint-defined manifold computing*, `cdmc()`, is a helper function, based on the method introduced in Gear et al. (2005*a*), that iteratively applies the microsolver and backward projection in time. The result is to project the fast variables close to the slow manifold, without advancing the current time by the burst time of the microsolver. This function reduces errors related to the simulation length of the microsolver in the `PIG` function. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

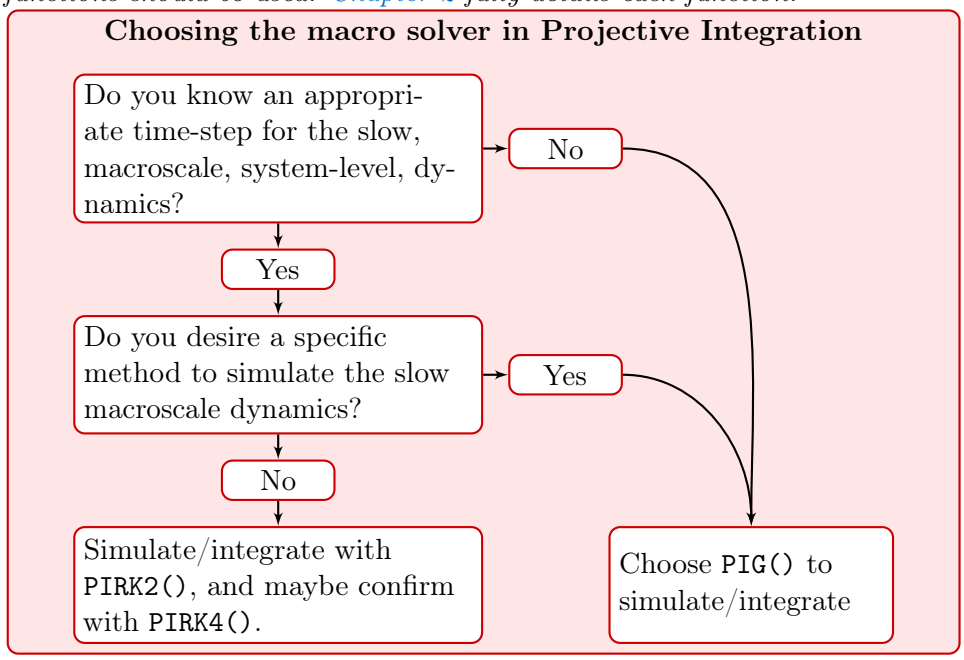
Figure 2.1: The Projective Integration method greatly accelerates simulation/ integration of a system exhibiting multiple time scales. The Projective Integration Chapter 2 presents several separate functions, as well as several optional wrapper functions that may be invoked. This chart overviews constructing a Projective Integration simulation, whereas Figure 2.2 roughly guides which top-level Projective Integration functions should be used. Chapter 2 fully details each function.



The above functions share dependence on a user-specified *microsolver* that accurately simulates some problem of interest.

The following sections describe the `PIRK2()` and `PIG()` functions in detail, providing an example for each. The function `PIRK4()` is very similar to `PIRK2()`. Descriptions for the minor functions follow, and an example using `cdmc()`.

Figure 2.2: The Projective Integration method greatly accelerates simulation/integration of a system exhibiting multiple time scales. In conjunction with Figure 2.1, this chart roughly guides which top-level Projective Integration functions should be used. Chapter 2 fully details each function.



2.2 PIRK2(): projective integration of second-order accuracy

Section contents

2.2.1 Introduction

This Projective Integration scheme implements a macroscale scheme that is analogous to the second-order Runge–Kutta Improved Euler integration.

```
21 function [x, tms, xms, rm, svf] = PIRK2(microBurst, tSpan, x0, bT)
```

Input If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in Section 2.2.2 as a basic template of how to use.

- `microBurst()`, a user-coded function that computes a short-time burst of the microscale simulation.

  `[tOut, xOut] = microBurst(tStart, xStart, bT)`

  - Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row n-vector of the starting state; `bT`, *optional*, the total time to simulate in the burst—if your `microBurst()` determines the burst time, then replace `bT` in the argument list by `varargin`.
  - Outputs: `tOut`, the column vector of solution times; and `xOut`, an array in which each *row* contains the system state at corresponding times.

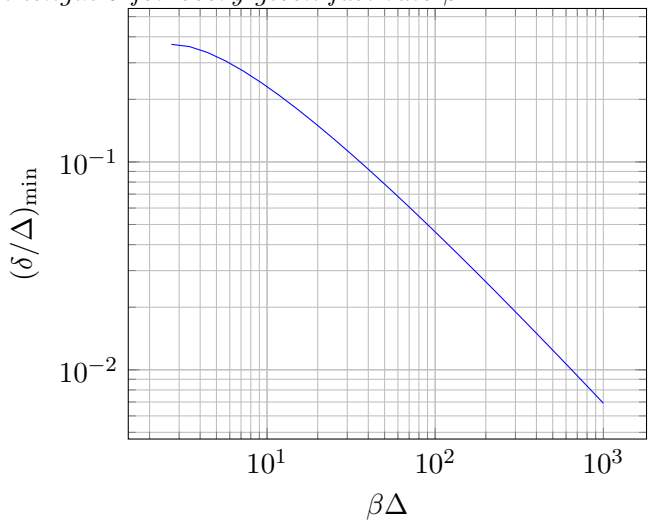
  Be wary that for very large scale separations (such as `MMepsilon<1e-5` in the Michaelis–Menten example), microscale integration by error-controlled variable-step routines (such as `ode23/45`) often generate microscale variations that ruin the projective extrapolation of `PIRK2()`. In such cases, a fixed time-step microscale integrator is much better (such as `rk2Int()`).

- `tSpan` is an ℓ-vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not use adaptive time-stepping; the macroscale time-steps are (nearly) the steps between elements of `tSpan`.

- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: such `Nan`s are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.

- `bT`, *optional*, either missing, or empty (`[]`), or a scalar: if a given scalar, then it is the length of the micro-burst simulations—the minimum amount of time needed for the microscale simulation to relax to the

Figure 2.3: Need macroscale step Δ such that $|\alpha\Delta| \lesssim \sqrt{6\varepsilon}$ for given relative error ε and slow rate α, and then $\delta/\Delta \gtrsim \frac{1}{\beta\Delta}\log|\beta\Delta|$ determines the minimum required burst length δ for every given fast rate β.



slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a burst.

```
77  if nargin<4, bT=[]; end
```

Choose a long enough burst length Suppose: firstly, you have some desired relative accuracy ε that you wish to achieve (e.g., $\varepsilon \approx 0.01$ for two digit accuracy); secondly, the slow dynamics of your system occurs at rate/frequency of magnitude about α; and thirdly, the rate of *decay* of your fast modes are faster than the lower bound β (e.g., if three fast modes decay roughly like $e^{-12t}, e^{-34t}, e^{-56t}$ then $\beta \approx 12$). Then set

1. a macroscale time-step, $\Delta =$ `diff(tSpan)`, such that $\alpha\Delta \approx \sqrt{6\varepsilon}$, and
2. a microscale burst length, $\delta =$ `bT` $\gtrsim \frac{1}{\beta}\log|\beta\Delta|$, see Figure 2.3.

Output If there are no output arguments specified, then a plot is drawn of the computed solution `x` versus `tSpan`.

- `x`, an $\ell \times n$ array of the approximate solution vector. Each row is an estimated state at the corresponding time in `tSpan`. The simplest usage is then `x = PIRK2(microBurst,tSpan,x0,bT)`.

  However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK2()` provides up to four optional outputs of the microscale bursts.

- `tms`, optional, is an L dimensional column vector containing the microscale times within the burst simulations, each burst separated by `NaN`;

- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding

time `tms` and helps visualise details of the solution.

- `rm`, optional, a struct containing the 'remaining' applications of the microBurst required by the Projective Integration method during the calculation of the macrostep:
  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; the `rm.x` are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and do *not* accurately approximate the macroscale dynamics.
- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.
  - `svf.t` is a 2ℓ dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along microBurst data to form a macrostep.
  - `svf.dx` is a $2\ell \times n$ array containing the estimated slow vector field.

2.2.2 If no arguments, then execute an example

```
182 if nargin==0
```

Example code for Michaelis–Menton dynamics The Michaelis–Menten enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$:

$$\frac{dx}{dt} = -x + (x + \tfrac{1}{2})y \quad \text{and} \quad \frac{dy}{dt} = \frac{1}{\epsilon}\big[x - (x+1)y\big]$$

(encoded in function `MMburst()` in the next paragraph). With initial conditions $x(0) = 1$ and $y(0) = 0$, the following code computes and plots a solution over time $0 \le t \le 6$ for parameter $\epsilon = 0.05$. Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(\Delta/\epsilon)$ as here the macroscale time-step $\Delta = 1$.

```
203 global MMepsilon
204 MMepsilon = 0.05
205 ts = 0:6
206 bT = MMepsilon*log( (ts(2)-ts(1))/MMepsilon )
207 [x,tms,xms] = PIRK2(@MMburst, ts, [1;0], bT);
208 figure, plot(ts,x,'o:',tms,xms)
209 title('Projective integration of Michaelis--Menten enzyme kinetics')
210 xlabel('time t'), legend('x(t)','y(t)')
```

Upon finishing execution of the example, exit this function.

```
216 return
217 end%if no arguments
```

Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), we here simply use MATLAB/Octave's `ode23`/`lsode` to integrate a burst in time.

```
15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
18         1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
19     if ~exist('OCTAVE_VERSION','builtin')
20     [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21     else % octave version
22     [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23     end
24 end
```

```
 8 function [ts,xs] = odeOct(dxdt,tSpan,x0)
 9     if length(tSpan)>2, ts = tSpan;
10     else ts = linspace(tSpan(1),tSpan(end),21);
11     end
12     % mimic ode45 and ode23, but much slower for non-PI
13     lsode_options('integration method','non-stiff');
14     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
15 end
```

2.3 egPIMM: Example projective integration of Michaelis–Menton kinetics

The Michaelis–Menten enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$:

$$\frac{dx}{dt} = -x + (x + \tfrac{1}{2})y \quad \text{and} \quad \frac{dy}{dt} = \frac{1}{\epsilon}\big[x - (x+1)y\big]$$

(encoded in function `MMburst()` below). As illustrated by Figure 2.5, the slow variable $x(t)$ evolves on a time scale of one, whereas the fast variable $y(t)$ evolves on a time scale of the small parameter ϵ.

Invoke projective integration Clear, and set the scale separation parameter ϵ to something small like 0.01. Here use $\epsilon = 0.1$ for clearer graphs.

```
clear all, close all
global MMepsilon
MMepsilon = 0.1
```

First, the end of this section encodes the computation of bursts of the Michaelis–Menten system in a function `MMburst()`. Second, here set macroscale times of computation and interest into vector `ts`. Then, invoke Projective Integration with `PIRK2()` applied to the burst function, say using bursts of simulations of length 2ϵ, and starting from the initial condition for the Michaelis–Menten system, at time $t = 0$, of $(x, y) = (1, 0)$ (off the slow manifold).

```
ts = 0:6
xs = PIRK2(@MMburst, ts, [1;0], 2*MMepsilon)
plot(ts,xs,'o:')
xlabel('time t'), legend('x(t)','y(t)')
title('macroscale points only')
ifOurCf2eps([mfilename '1'])
pause(1)
```

Figure 2.4 plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold. Sieber et al. (2018) [§4] used this system as an example of their analysis of the convergence of Projective Integration.

Request and plot the microscale bursts Because the initial conditions of the simulation are off the slow manifold, the initial macroscale step appears to 'jump' (Figure 2.4). In order to see the initial transient attraction to the slow manifold we plot some microscale data in Figure 2.5. Two further output variables provide this microscale burst information.

```
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, [1;0], 2*MMepsilon);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
title('macroscale points with microscale bursts')
ifOurCf2eps([mfilename '2'])
pause(1)
```

Figure 2.4: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`: macroscale samples.

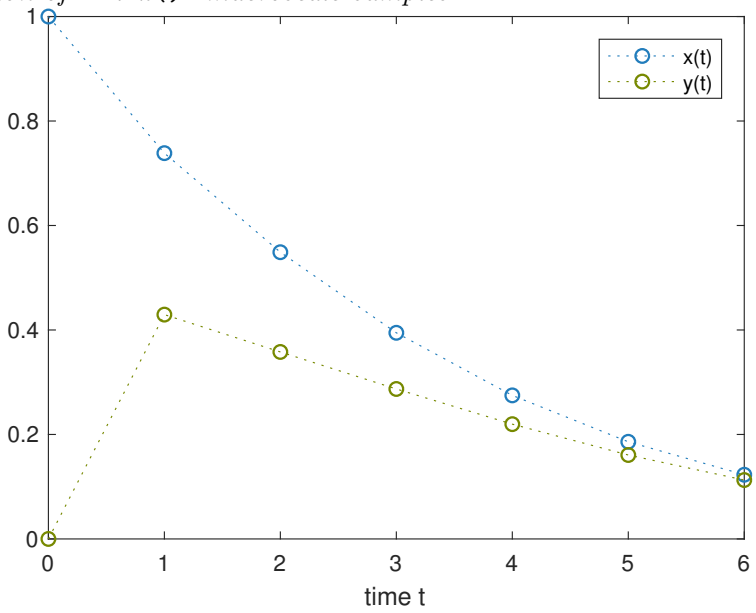


Figure 2.5 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable $x(t)$ is also affected by the initial transient (hence other schemes which 'freeze' slow variables are less accurate).

Simulate backward in time Figure 2.6 shows that projective integration even simulates backward in time along the slow manifold using short forward bursts (Gear & Kevrekidis 2003*a*, Frewen et al. 2009). Such backward macroscale simulations succeed despite the fast variable $y(t)$, when backward in time, being viciously unstable. However, backward integration appears to need longer bursts, here 3ϵ.

```
ts = 0:-1:-5
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, 0.2*[1;1], 3*MMepsilon);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
title('backward integration showing points with bursts')
ifOurCf2eps([mfilename '3'])
```

Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), we here simply use MATLAB/Octave's `ode23`/`lsode` to integrate a burst in time.

```
function [ts, xs] = MMburst(ti, xi, bT)
    global MMepsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
        1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
```

Figure 2.5: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`*: the microscale bursts show the initial transients on a time scale of* $\epsilon = 0.1$*, and then the alignment along the slow manifold.*

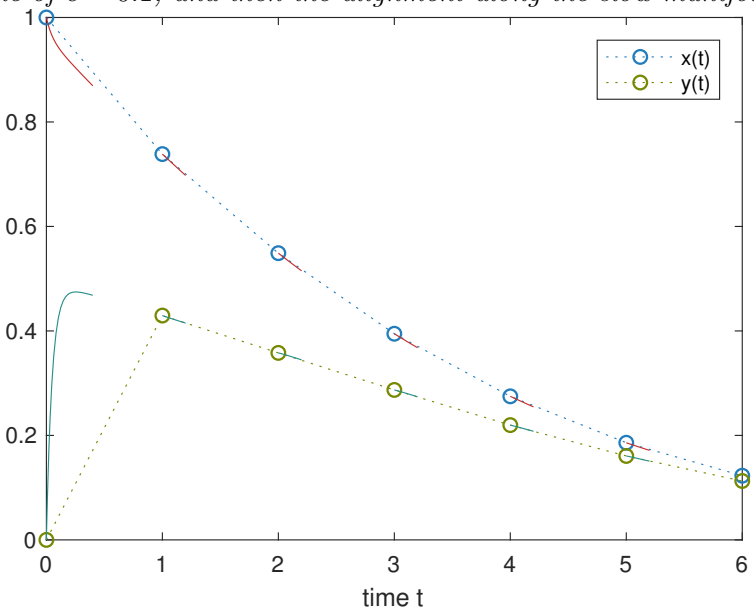
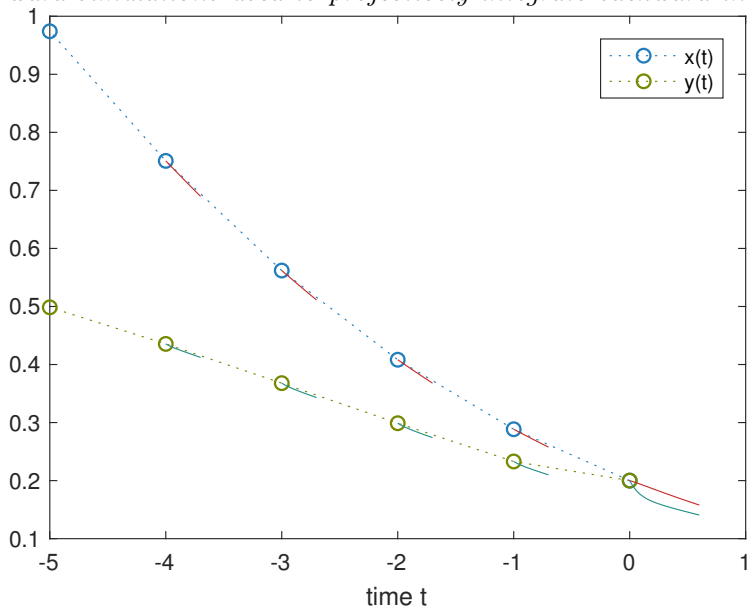


Figure 2.6: Michaelis–Menten enzyme kinetics at $\epsilon = 0.1$ *simulated backward with the projective integration of* `PIRK2()`*: the microscale bursts show the short forward simulations used to projectively integrate backward in time.*



```
    if ~exist('OCTAVE_VERSION','builtin')
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
    else % octave version
    [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
    end
end
```

```
function [ts,xs] = odeOct(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21);
    end
    % mimic ode45 and ode23, but much slower for non-PI
    lsode_options('integration method','non-stiff');
    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
end
```

2.4 PIG(): Projective Integration via a General macroscale integrator

Section contents

2.4.1 Introduction

This is a Projective Integration scheme when the macroscale integrator is any specified coded method. The advantage is that one may use MATLAB/Octave's inbuilt integration functions, with all their sophisticated error control and adaptive time-stepping, to do the macroscale integration/simulation.

By default, for the microscale simulations `PIG()` uses 'constraint-defined manifold computing', `cdmc()` (Section 2.6). This algorithm, initiated by Gear et al. (2005*b*), uses a backward projection so that the simulation time is unchanged after running the microscale simulator.

```
30 function [T,X,tms,xms,svf] = PIG(macroInt,microBurst,Tspan,x0 ...
31                                  ,restrict,lift,cdmcFlag)
```

Inputs:

- `macroInt()`, the numerical method that the user wants to apply on a slow-time macroscale. Either specify a standard MATLAB/Octave integration function (such as `'ode23'` or `'ode45'`), or code your own integration function using standard arguments. That is, if you code your own, then it must be

  ```
  [Ts,Xs] = macroInt(F,Tspan,X0)
  ```

  where

  - function `F(T,X)` notionally evaluates the time derivatives $d\vec{X}/dt$ at any time;
  - `Tspan` is either the macro-time interval, or the vector of macroscale times at which macroscale values are to be returned; and
  - `X0` are the initial values of $\vec{X}$ at time `Tspan(1)`.

  Then the ith *row* of `Xs`, `Xs(i,:)`, is to be the vector $\vec{X}(t)$ at time $t =$ `Ts(i)`. Remember that in `PIG()` the function `F(T,X)` is to be estimated by Projective Integration.

- `microBurst()` is a function that produces output from the user-specified code for a burst of microscale simulation. The function must internally specify/decide how long a burst it is to use. Usage

  ```
  [tbs,xbs] = microBurst(tb0,xb0)
  ```

Inputs: `tb0` is the start time of a burst; `xb0` is the n-vector microscale state at the start of a burst.

Outputs: `tbs`, the vector of solution times; and `xbs`, the corresponding microscale states.

- `Tspan`, a vector of macroscale times at which the user requests output. The first element is always the initial time. If `macroInt` reports adaptively selected time steps (e.g., `ode45`), then `Tspan` consists of an initial and final time only.
- `x0`, the n-vector of initial microscale values at the initial time `Tspan(1)`.

Optional Inputs: `PIG()` allows for none, two or three additional inputs after `x0`. If you distinguish distinct microscale and macroscale states and your aim is to do Projective Integration on the macroscale only, then lifting and restriction functions must be provided to convert between them. Usage `PIG(...,restrict,lift)`:

- `restrict(x)`, a function that takes an input high-dimensional, n-D, microscale state $\vec{x}$ and computes the corresponding low-dimensional, N-D, macroscale state $\vec{X}$;
- `lift(X,xApprox)`, a function that converts an input low-dimensional, N-D, macroscale state $\vec{X}$ to a corresponding high-dimensional, n-D, microscale state $\vec{x}$, given that `xApprox` is a recently computed microscale state on the slow manifold.

Either both `restrict()` and `lift()` are to be defined, or neither. If neither are defined, then they are assumed to be identity functions, so that `N=n` in the following.

If desired, the default constraint-defined manifold computing microsolver may be disabled, via `PIG(...,restrict,lift,cdmcFlag)`

- `cdmcFlag`, *any* seventh input to `PIG()`, will disable `cdmc()`, e.g., the string `'cdmc off'`.

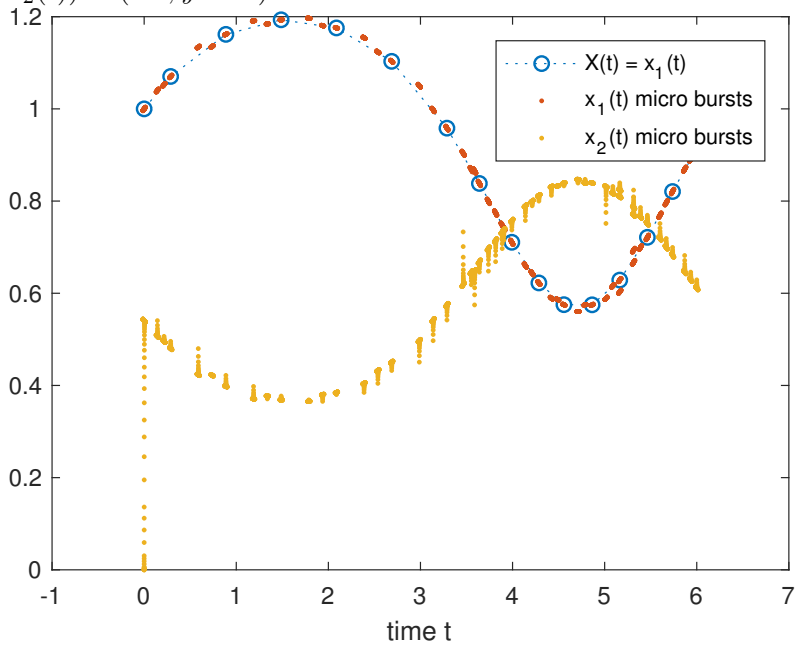
If the `cdmcFlag` is to be set without using a `restrict()` or `lift()` function, then use empty matrices `[]` for the restrict and lift functions.

Output Between zero and five outputs may be requested. If there are no output arguments specified, then a plot is drawn of the computed solution `X` versus `T`. Most often you would store the first two output results of `PIG()`, via say `[T,X] = PIG(...)`.

- `T`, an L-vector of times at which `macroInt` produced results.
- `X`, an $L \times N$ array of the computed solution: the ith *row* of `X`, `X(i,:)`, is to be the macro-state vector $\vec{X}(t)$ at time $t =$ `T(i)`.

However, microscale details of the underlying Projective Integration computations may be helpful, and so `PIG()` provides some optional outputs of the microscale bursts, via `[T,X,tms,xms] = PIG(...)`

Figure 2.7: Projective Integration by `PIG` *of the example system (2.1) with* $\epsilon = 10^{-3}$ *(Section 2.4.2). The macroscale solution* $X(t)$ *is represented by just the blue circles. The microscale bursts are the microscale states* $(x_1(t), x_2(t)) = (red, yellow)$ *dots.*



- `tms`, optional, is an ℓ-dimensional column vector containing microscale times with bursts, each burst separated by `NaN`;
- `xms`, optional, is an $\ell \times n$ array of the corresponding microscale states.

In some contexts it may be helpful to see directly how Projective Integration approximates a reduced slow vector field, via `[T,X,tms,xms,svf] = PIG(...)` in which

- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.
  - `svf.T` is a $\hat{L}$-dimensional column vector containing all times at which the microscale simulation data is extrapolated to form an estimate of $d\vec{x}/dt$ in `macroInt()`.
  - `svf.dX` is a $\hat{L} \times N$ array containing the estimated slow vector field.

If `macroInt()` is, for example, the forward Euler method (or the Runge–Kutta method), then $\hat{L} = L$ (or $\hat{L} = 4L$).

2.4.2 If no arguments, then execute an example

```
180 if nargin==0
```

As a basic example, consider a microscale system of the singularly perturbed system of differential equations

$$\frac{dx_1}{dt} = \cos(x_1)\sin(x_2)\cos(t) \quad \text{and} \quad \frac{dx_2}{dt} = \frac{1}{\epsilon}\big[\cos(x_1) - x_2\big]. \tag{2.1}$$

The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and plots a solution of the system (2.1) over time $0 \le t \le 6$ for parameter $\epsilon = 10^{-3}$(Figure 2.7). Whenever needed by `microBurst()`, the microscale system (2.1) is initialised ('`lifted`') using $x_2(t) = x_2^{\text{approx}}$ (yellow dots in Figure 2.7).

First we code the right-hand side function of the microscale system (2.1) of ODEs.

```
epsilon = 1e-3;
dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
              ( cos(x(1))-x(2) )/epsilon ];
```

Second, we code microscale bursts, here using the standard `ode45()`. We choose a burst length $2\epsilon\log(1/\epsilon)$ as the rate of decay is $\beta \approx 1/\epsilon$ but we do not know the macroscale time-step invoked by `macroInt()`, so blithely assume $\Delta \le 1$ and then double the usual formula for safety.

```
bT = 2*epsilon*log(1/epsilon)
if ~exist('OCTAVE_VERSION','builtin')
    micB='ode45'; else micB='rk2Int'; end
microBurst = @(tb0, xb0) feval(micB,dxdt,[tb0 tb0+bT],xb0);
```

Third, code functions to convert between macroscale and microscale states.

```
restrict = @(x) x(1);
lift = @(X,xApprox) [X; xApprox(2)];
```

Fourth, invoke `PIG` to use MATLAB/Octave's `ode23`/`lsode`, say, on the macroscale slow evolution. Integrate the micro-bursts over $0 \le t \le 6$ from initial condition $\vec{x}(0) = (1, 0)$. You could set `Tspan=[0 -6]` to integrate backward in macroscale time with forward microscale bursts (Gear & Kevrekidis 2003*a*, Frewen et al. 2009).

```
Tspan = [0 6];
x0 = [1;0];
if ~exist('OCTAVE_VERSION','builtin')
    macInt='ode23'; else macInt='odeOct'; end
[Ts,Xs,tms,xms] = PIG(macInt,microBurst,Tspan,x0,restrict,lift);
```

Plot output of this projective integration.

```
figure, plot(Ts,Xs,'o:',tms,xms,'.')
title('Projective integration of singularly perturbed ODE')
xlabel('time t')
legend('X(t) = x_1(t)','x_1(t) micro bursts','x_2(t) micro bursts')
```

Upon finishing execution of the example, exit this function.

```
return
end%if no arguments
```

2.5 PIRK4(): projective integration of fourth-order accuracy

Section contents

2.5.1 Introduction

This Projective Integration scheme implements a macrosolver analogous to the fourth-order Runge–Kutta method.

```
19 function [x, tms, xms, rm, svf] = PIRK4(microBurst, tSpan, x0, bT)
```

See Section 2.2 as the inputs and outputs are the same as `PIRK2()`.

If no arguments, then execute an example

```
29 if nargin==0
```

Example of Michaelis–Menton backwards in time The Michaelis–Menten enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$ (encoded in function `MMburst`):

$$\frac{dx}{dt} = -x + (x + \tfrac{1}{2})y \quad \text{and} \quad \frac{dy}{dt} = \frac{1}{\epsilon}\big[x - (x+1)y\big].$$

With initial conditions $x(0) = y(0) = 0.2$, the following code uses forward time bursts in order to integrate backwards in time to $t = -5$ (Frewen et al. 2009, e.g.). It plots the computed solution over time $-5 \le t \le 0$ for parameter $\epsilon = 0.1$. Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(|\Delta|/\epsilon)$ as here the macroscale time-step $\Delta = -1$.

```
50 global MMepsilon
51 MMepsilon = 0.1
52 ts = 0:-1:-5
53 bT = MMepsilon*log(abs(ts(2)-ts(1))/MMepsilon)
54 [x,tms,xms,rm,svf] = PIRK4(@MMburst, ts, 0.2*[1;1], bT);
55 figure, plot(ts,x,'o:',tms,xms)
56 xlabel('time t'), legend('x(t)','y(t)')
57 title('Backwards-time projective integration of Michaelis--Menten')
```

Upon finishing execution of the example, exit this function.

```
63 return
64 end%if no arguments
```

Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), we here simply use MATLAB/Octave's `ode23`/`lsode` to integrate a burst in time.

```
15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
18         1/MMepsilon*( x(1)-(x(1)+1)*x(2) ) ];
19     if ~exist('OCTAVE_VERSION','builtin')
20     [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21     else % octave version
22     [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23     end
24 end
```

```
 8 function [ts,xs] = odeOct(dxdt,tSpan,x0)
 9     if length(tSpan)>2, ts = tSpan;
10     else ts = linspace(tSpan(1),tSpan(end),21);
11     end
12     % mimic ode45 and ode23, but much slower for non-PI
13     lsode_options('integration method','non-stiff');
14     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
15 end
```

2.6 cdmc(): constraint defined manifold computing

The function cdmc() iteratively applies the given micro-burst and then projects backward to the initial time. The cumulative effect is to relax the variables to the attracting slow manifold, while keeping the 'final' time for the output the same as the input time.

```
17 function [ts, xs] = cdmc(microBurst, t0, x0)
```

Input

- microBurst(), a black-box micro-burst function suitable for Projective Integration. See any of PIRK2(), PIRK4(), or PIG() for a description of microBurst().
- t0, an initial time.
- x0, an initial state vector.

Output

- ts, a vector of times.
- xs, an array of state estimates produced by microBurst().

This function is a wrapper for the micro-burst. For instance if the problem of interest is a dynamical system that is not too stiff, and which is simulated by the micro-burst function sol(t,x), one would invoke cdmc() by defining

```
cdmcSol = @(t,x) cdmc(sol,t,x)|
```

and thereafter use cdmcSol() in place of sol() as the microBurst in any Projective Integration scheme. The original microBurst sol() could create large errors if used in the PIG() scheme, but the output via cdmc() should not.

3 Patch scheme for given microscale discrete space system

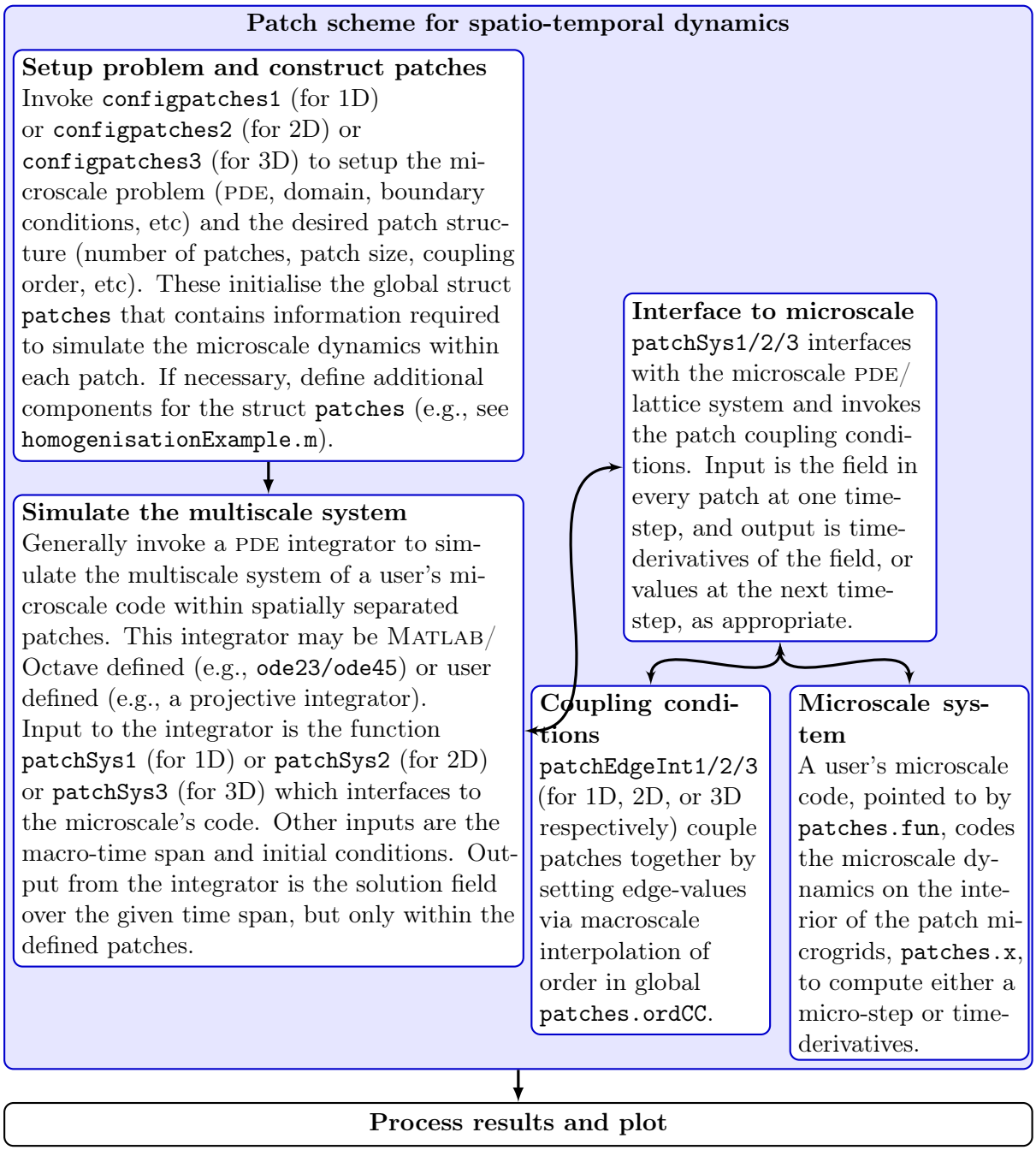
Chapter contents

Consider spatio-temporal multiscale systems where the spatial domain is so large that a given microscale code cannot be computed in a reasonable time. The *patch scheme* computes the microscale details only on small patches of the space-time domain, and produce correct macroscale predictions by craftily coupling the patches across unsimulated space (Hyman 2005, Samaey et al. 2005, 2006, Roberts & Kevrekidis 2007, Liu et al. 2015, e.g.). The resulting macroscale predictions were generally proved to be consistent with the microscale dynamics, to some specified order of accuracy, in a series of papers: 1D-space dissipative systems (Roberts & Kevrekidis 2007, Bunder et al. 2017); 2D-space dissipative systems (Roberts et al. 2014, Bunder et al. 2020); and 1D-space wave-like systems (Cao & Roberts 2016).

The microscale spatial structure is to be on a lattice such as obtained from finite difference/element/volume approximation of a PDE. The microscale is either continuous or discrete in time.

Quick start See Sections 3.1.1 and 4.1.1 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

Figure 3.1: The Patch methods, Chapter 3, accelerate simulation/integration of multiscale systems with interesting spatial/network structure/patterns. The methods use your given microsimulators whether coded from PDE*s, lattice systems, or agent/particle microscale simulators. The patch functions require that a user configure the patches, and interface the coupled patches with a time integrator/simulator. This chart overviews the main functional recursion involved.*



3.1 `configPatches1()`: configure spatial patches in 1D

Section contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys1()`. Section 3.1.1 lists an example of its use.

```
19 function patches = configPatches1(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-03-23';
```

Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 3.1.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D patches.
- `Xlim` give the macro-space spatial domain of the computation, namely the interval [`Xlim(1)`, `Xlim(2)`].
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is macro-periodic in the 1D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
  - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left(right) edge of the leftmost(rightmost) patches.
  - `.bcOffset`, optional one or two element array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right edges of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when applying Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when applying Neumann boundary conditions halfway between the extreme edge micro-grid points.
  - `.X`, optional array, in the case `'usergiven'` it specifies the locations of the centres of the `nPatch` patches—the user is responsible it makes sense.
- `nPatch` is the number of equi-spaced spatial patches.

- `ordCC`, must be ≥ -1, is the 'order' of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd specifies staggered spatial grids.
- `dx` (real) is usually the sub-patch micro-grid spacing in x.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio`, namely the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the equi-spacing of the patch mid-points—adjusted a little when `nEdge` > 1. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. If not using `EdgyInt`, then `nSubP/nEdge` must be odd integer so that there is/are centre-patch lattice point(s). So for the defaults of `nEdge` $= 1$ and not `EdgyInt`, then `nSubP` must be odd.
- `'nEdge'`, *optional*, default=1, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `hetCoeffs`, *optional*, default empty. Supply a 1D or 2D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times n_c$, where n_c is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of `(nSubP-nEdge)/2` for not `EdgyInt`.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x$ and construct an ensemble of all m_x phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry.
- `nCore`, *optional-experimental*, default one, but if more, and only for non-EdgyInt, then interpolates from an average over the core of a patch,

a core of size ??. Then edge values are set according to interpolation of the averages?? or so that average at edges is the interpolant??

- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x. A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[1]

```
187 if nargout==0, global patches, end
188 patches.version = version;
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- `.Cwtsr` and `.Cwtsl`, only for macro-periodic conditions, are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified or as derived from `dx`.
- `.x` (4D) is `nSubP`$\times 1 \times 1 \times$`nPatch` array of the regular spatial locations x_{iI} of the ith microscale grid point in the Ith patch.
- `.ratio`, only for macro-periodic conditions, is the size ratio of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.

[1] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1) \times n_c \times m_x$ 3D array of m_x ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

3.1.1 If no arguments, then execute an example

```
261 if nargin==0
262 disp('With no arguments, simulate example of Burgers PDE')
```

The code here shows one way to get started: a user's script may have the following three steps (" $\mapsto$ " denotes function recursion).

1. configPatches1
2. ode15s integrator $\mapsto$ patchSys1 $\mapsto$ user's PDE
3. process results

Establish global patch data struct to point to and interface with a function coding Burgers' PDE: to be solved on 2π-periodic domain, with eight patches, spectral interpolation couples the patches, with micro-grid spacing 0.06, and with seven microscale points forming each patch.

```
282 global patches
283 patches = configPatches1(@BurgersPDE, [0 2*pi], ...
284     'periodic', 8, 0, 0.06, 7);
```

Set some initial condition, with some microscale randomness.

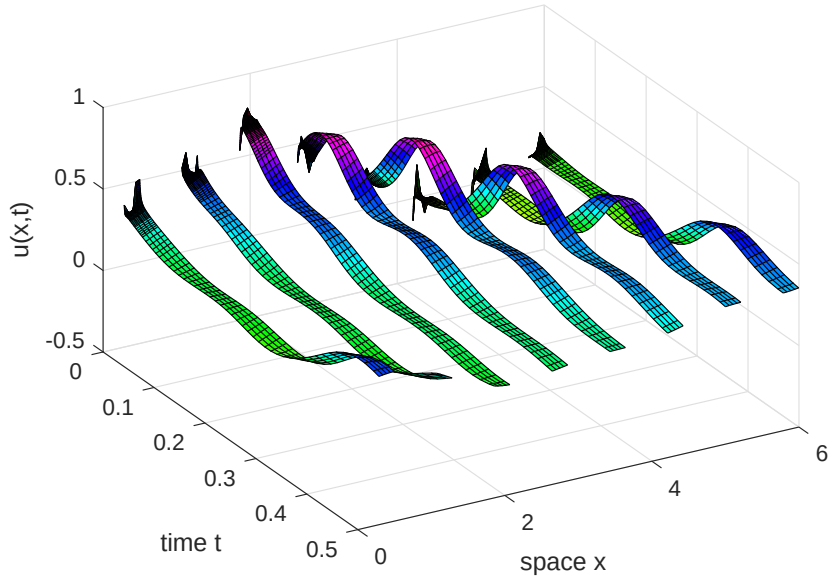
```
290 u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
```

Simulate in time using a standard stiff integrator and the interface function `patchSys1()` (Section 3.2).

```
298 if ~exist('OCTAVE_VERSION','builtin')
299 [ts,us] = ode15s( @patchSys1,[0 0.5],u0(:));
300 else % octave version
301 [ts,us] = odeOcts(@patchSys1,[0 0.5],u0(:));
302 end
```

Plot the simulation using only the microscale values interior to the patches: either set x-edges to `nan` to leave the gaps; or use `patchEdgyInt1` to re-interpolate correct patch edge values and thereby join the patches. Figure 3.2 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

Figure 3.2: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.




```
figure(1),clf
if 1, patches.x([1 end],:,:,:)=nan;  us=us.';
else us=reshape(patchEdgyInt1(us.'),[],length(ts));
end
mesh(ts,patches.x(:),us)
view(60,40), colormap(0.7*hsv)
title('Burgers PDE: patches in space, continuous time')
xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
```

Upon finishing execution of the example, optionally save the graph to be shown in Figure 3.2, then exit this function.

```
ifOurCf2eps(mfilename)
return
end%if nargin==0
```

Example of Burgers PDE inside patches As a microscale discretisation of Burgers' PDE $u_t = u_{xx} - 30uu_x$, here code $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$. Here there is only one field variable, and one in the ensemble, so for simpler coding of the PDE we squeeze them out (with no need to reshape when via patchSys1()).

```
function ut=BurgersPDE(t,u,patches)
  u=squeeze(u);     % omit singleton dimensions
  dx=diff(patches.x(1:2));  % microscale spacing
  i=2:size(u,1)-1;  % interior points in patches
  ut=nan+u;         % preallocate output array
  ut(i,:)=diff(u,2)/dx^2 ...
    -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
end
```

```
10 function [ts,xs] = odeOcts(dxdt,tSpan,x0)
11     if length(tSpan)>2, ts = tSpan;
12     else ts = linspace(tSpan(1),tSpan(end),21)';
13     end
14     lsode_options('integration method','non-stiff');
15     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16 end
```

3.2 patchSys1(): interface 1D space to time integrators

To simulate in time with 1D spatial patches we often need to interface a user's time derivative function with time integration routines such as ode23 or PIRK2. This function provides an interface. Communicate patch-design variables (Section 3.1) either via the global struct patches or via an optional third argument. patches is required for the parallel computing of spmd, or if parameters are to be passed though to the user microscale function.

```
23 function dudt=patchSys1(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

Input

- u is a vector/array of length nSubP · nVars · nEnsem · nPatch where there are nVars·nEnsem field values at each of the points in the nSubP × nPatch grid.
- t is the current time to be passed to the user's time derivative function.
- patches a struct set by configPatches1() with the following information used here.
  - .fun is the name of the user's function fun(t,u,patches,...) that computes the time derivatives on the patchy lattice. The array u has size nSubP × nVars × nEnsem × nPatch. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - .x is nSubP × 1 × 1 × nPatch array of the spatial locations x_i of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale.
- varargin, optional, is arbitrary number of parameters to be passed onto the users time-derivative function as specified in configPatches1.

Output

- dudt is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length nSubP · nVars · nEnsem · nPatch and the same dimensions as u.

3.3 `patchEdgeInt1()`: sets patch-edge values from interpolation over the 1D macroscale

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value (Roberts 2003, Roberts & Kevrekidis 2007), or the patch-core average (Bunder et al. 2017), or the opposite next-to-edge values (Bunder et al. 2020)—this last alternative often maintains symmetry. This function is primarily used by `patchSys1()` but is also useful for user graphics. When using core averages (not fully implemented), assumes the averages are sensible macroscale variables: then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). [2]

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`), or otherwise via the global struct `patches`.

```
31 function u=patchEdgeInt1(u,patches)
32 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length $\texttt{nSubP}\cdot\texttt{nVars}\cdot\texttt{nEnsem}\cdot\texttt{nPatch}$ where there are $\texttt{nVars}\cdot\texttt{nEnsem}$ field values at each of the points in the $\texttt{nSubP}\times\texttt{nPatch}$ multiscale spatial grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is $\texttt{nSubP}\times 1\times 1\times\texttt{nPatch}$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.ordCC` is order of interpolation, integer ≥ -1.
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left and right boundaries so interpolation is via divided differences.
  - `.stag` in $\{0,1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation—when invoking a periodic domain.
  - `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre-patch values (original scheme).
  - `.nEdge`, for each patch, the number of edge values set by interpolation at the edge regions of each patch (default is one).

[2] Script `patchEdgeInt1test.m` verifies this code.

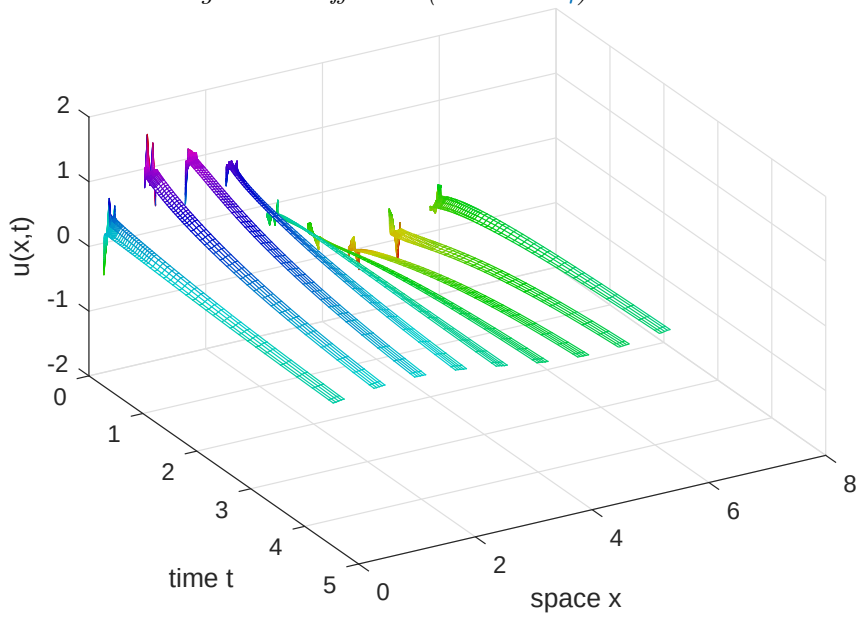
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.
- `.nCore` [3]

Output

- `u` is 4D array, `nSubP` × `nVars` × `nEnsem` × `nPatch`, of the fields with edge values set by interpolation.

[3] **ToDo:** introduced sometime but not fully implemented yet, because prefer ensemble

Figure 3.3: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion (Section 3.4).



3.4 `homogenisationExample`: simulate heterogeneous diffusion in 1D on patches

Section contents

Figure 3.3 shows an example simulation in time generated by the patch scheme applied to heterogeneous diffusion. That such simulations of heterogeneous diffusion makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is one more than a multiple of the periodic of the microscale heterogeneity.

The first part of the script implements the following gap-tooth scheme (left-right arrows denote function recursion).

1. configPatches1
2. ode15s $\mapsto$ patchSys1 $\mapsto$ heteroDiff
3. process results

Consider a lattice of values $u_i(t)$, with lattice spacing dx, and governed by the heterogeneous diffusion

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \tag{3.1}$$

In this 1D space, the macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients.

3.4.1 Script to simulate via stiff or projective integration

Set the desired microscale periodicity, and correspondingly choose random microscale diffusion coefficients (with subscripts shifted by a half).

```
mPeriod = 3
cDiff = exp(randn(mPeriod,1))
cHomo = 1/mean(1./cDiff)
```

Establish global data struct `patches` for heterogeneous diffusion on 2π-periodic domain. Use nine patches, each patch of half-size ratio 0.2. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Here include the diffusivity coefficients, repeated to fill up a patch.

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 2*mPeriod+1
Len = 2*pi;
ordCC = 4;
configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
    ,ordCC,ratio,nSubP,'hetCoeffs',cDiff);
```

For comparison: conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 3.2) to the microscale differential equations.

```
u0 = sin(patches.x)+0.3*randn(nSubP,1,1,nPatch);
if ~exist('OCTAVE_VERSION','builtin')
[ts,ucts] = ode15s(@patchSys1, [0 2/cHomo], u0(:));
else % octave version
[ts,ucts] = odeOcts(@patchSys1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

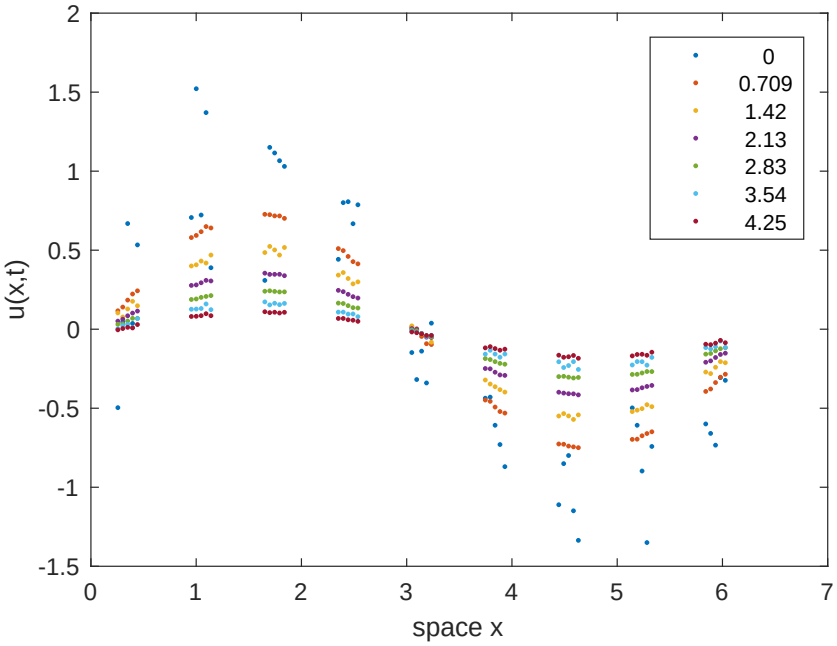
Plot the simulation in Figure 3.3.

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),ucts'),  view(60,40)
xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
ifOurCf2eps([mfilename 'CtsU'])
```

The code may invoke this integration interface.

```
function [ts,xs] = odeOcts(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21)';
    end
    lsode_options('integration method','non-stiff');
```

Figure 3.4: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion: different colours correspond to the times in the legend. This field solution displays some fine scale heterogeneity due to the heterogeneous diffusion.




```
15      xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16  end
```

Use projective integration in time Now take `patchSys1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.4.3), as illustrated by Figure 3.4.

This second part of the script implements the following design, where the micro-integrator could be, for example, `ode45` or `rk2int`.

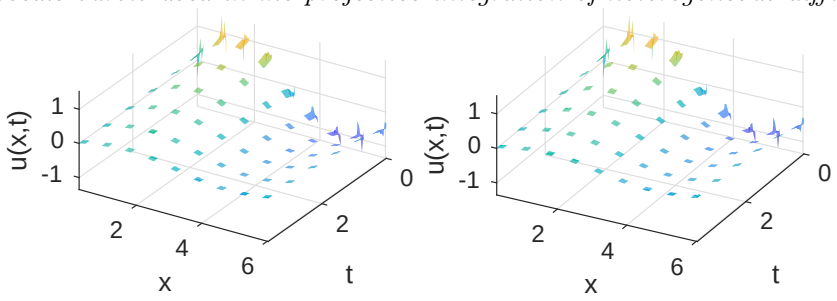
1. configPatches1 (done in first part)
2. PIRK2 $\mapsto$ heteroBurst $\mapsto$ micro-integrator $\mapsto$ patchSys1 $\mapsto$ heteroDiff
3. process results

Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
141 u0([1 end],:) = nan;
```

Set the desired macro- and microscale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the macroscale; the burst time is proportional to the intra-patch effective diffusion time; and lastly, the microscale time-step is proportional to the diffusion time between adjacent points in the microscale lattice.

Figure 3.5: cross-eyed stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration of heterogeneous diffusion.




```
ts = linspace(0,2/cHomo,7)
bT = 3*( ratio*Len/nPatch )^2/cHomo
addpath('../ProjInt')
[us,tss,uss] = PIRK2(@heteroBurst, ts, u0(:), bT);
```

Plot the macroscale predictions to draw Figure 3.4.

```
figure(2),clf
plot(xs(:),us','.')
ylabel('$u(x,t)$'), xlabel('space $x$')
legend(num2str(ts',3))
ifOurCf2eps([mfilename 'U'])
```

Also plot a surface detailing the microscale bursts as shown in the stereo Figure 3.5.

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('$x$'), xlabel('$t$'), zlabel('$u(x,t)$')
  axis tight, view(126-4*k,45)
end
ifOurCf2eps([mfilename 'Micro'])
```

End of this example script.

3.4.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSys1`, Section 3.2), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector of diffusivities c_i, and possibly Burgers' advection coefficients b_i, have previously been stored in struct `patches.cs`.

```
function ut = heteroDiff(t,u,patches)
  dx = diff(patches.x(2:3));   % space step
  i = 2:size(u,1)-1;   % interior points in a patch
  ut = nan+u;          % preallocate output array
  ut(i,:,:,:) = diff(patches.cs(:,1,:).*diff(u))/dx^2;
```

```
26    % possibly include heterogeneous Burgers' advection
27    if size(patches.cs,2)>1 % check for advection coeffs
28        buu = patches.cs(:,2,:).*u.^2;
29        ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
30    end
31  end% function
```

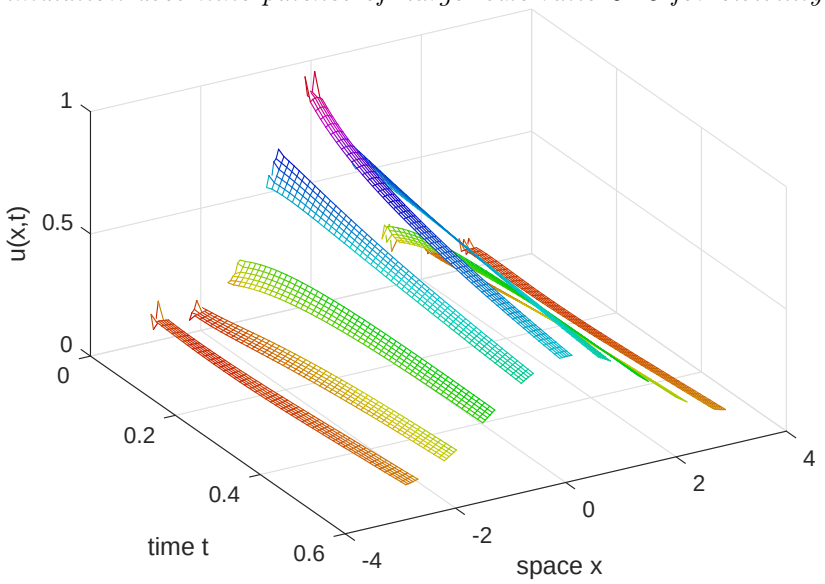
3.4.3 `heteroBurst()`: a burst of heterogeneous diffusion

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSys1`. Try `ode23` or `rk2Int`, although `ode45` may give smoother results.

```
15  function [ts, ucts] = heteroBurst(ti, ui, bT)
16          if ~exist('OCTAVE_VERSION','builtin')
17          [ts,ucts] = ode23( @patchSys1,[ti ti+bT],ui(:));
18          else % octave version
19          [ts,ucts] = rk2Int(@patchSys1,[ti ti+bT],ui(:));
20          end
21  end
```

Fin.

Figure 3.6: diffusion field $u(x,t)$ of the gap-tooth scheme applied to the diffusion (3.2). The microscale random component to the initial condition, the sub-patch fluctuations, decays, leaving the emergent macroscale diffusion. This simulation uses nine patches of 'large' size ratio 0.25 for visibility.



3.5 `homoDiffEdgy1`: computational homogenisation of a 1D heterogeneous diffusion by simulation on small patches

Figure 3.6 shows an example simulation in time generated by the patch scheme applied to macroscale diffusion propagation through a medium with microscale heterogeneity. The inter-patch coupling is realised by quartic interpolation of the patch's next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems, and quartic appears to be the lowest order that generally gives good accuracy.

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[c_{i-1/2}\delta u_i], \tag{3.2}$$

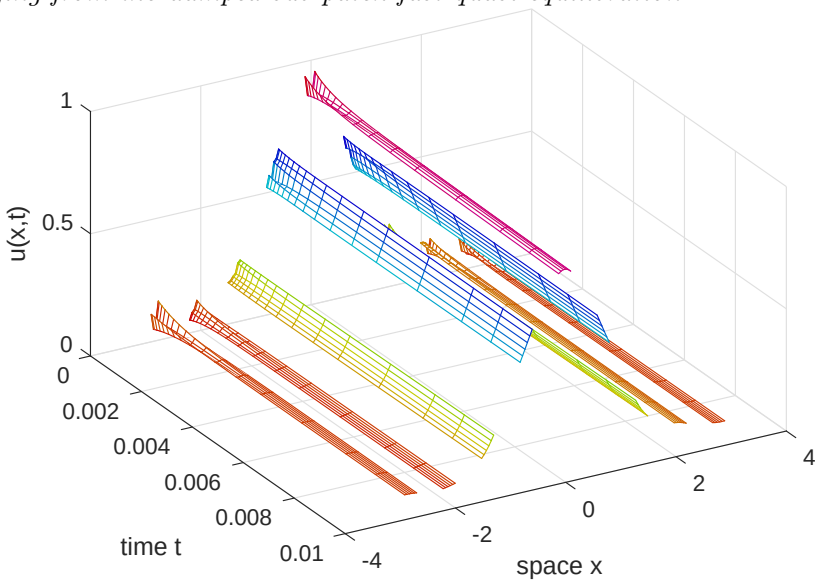
in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.6 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

3.5.1 Script code to simulate heterogeneous diffusion systems

This example script implements the following patch/gap-tooth scheme (left-right arrows denote function recursion).

1. configPatches1
2. ode15s $\mapsto$ patchSys1 $\mapsto$ heteroDiff
3. plot the simulation

Figure 3.7: diffusion field $u(x,t)$ of the gap-tooth scheme applied to the diffusive (3.2). Over this short meso-time we see the macroscale diffusion emerging from the damped sub-patch fast quasi-equilibration.



4. use patchSys1 to explore the Jacobian

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and random log-normal values, albeit normalised to have harmonic mean one. This normalisation then means that macroscale diffusion on a domain of length 2π should have near integer decay rates, the squares of $0, 1, 2, \ldots$. Then the heterogeneity is repeated to fill each patch, and phase-shifted for an ensemble.

```
mPeriod = 3%randi([2 5])
% set random diffusion coefficients
cHetr=exp(0.3*randn(mPeriod,1));
%cHetr = [3.966;2.531;0.838;0.331;7.276];
cHetr = cHetr*mean(1./cHetr) % normalise
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.2) solved on 2π-periodic domain, with seven patches, here each patch of size ratio 0.25 from one side to the other, with five micro-grid points in each patch, and quartic interpolation (4) to provide the edge-values of the inter-patch coupling conditions. Setting `patches.EdgyInt` to one means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values). In this case we appear to need at least fourth order (quartic) interpolation to get reasonable decay rate for heterogeneous diffusion. When simulating an ensemble of configurations, `nSubP` (the number of points in a patch) need not be dependent on the period of the heterogeneous diffusion.

```
global patches
nPatch = 9
ratio = 0.25;
```

```
nSubP = mPeriod+1 %randi([mPeriod+1 2*mPeriod+2])
nEnsem = mPeriod % number realisations in ensemble
if mod(nSubP,mPeriod)==2, nEnsem=1, end
configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
    ,4,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr);
```

Simulate Set the initial conditions of a simulation to be that of a lump perturbed by significant random microscale noise, via `randn`.

```
u0 = 0.8*exp(-patches.x.^2)+0.2*rand(nSubP,1,nEnsem,nPatch);
du0dt = patchSys1(0,u0(:));
```

Integrate using standard integrators.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys1, [0 0.6], u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSys1, 0.6*linspace(0,1).^2, u0(:));
end
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so we adjoin a row of `nan`s in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nan`s, and reshape again. In the case of an ensemble of phase-shifts, we plot the mean over the ensemble.

```
xs = squeeze(patches.x);
us = patchEdgeInt1( permute( reshape(us ...
    ,length(ts),nSubP,nEnsem,nPatch) ,[2 1 3 4]) );
ustd = squeeze(std(us,0,3));
us = squeeze(mean(us,3));
if 0, % omit interpolated edges
    us([1 end],:,:) = nan;
    ustd([1 end],:,:) = nan;
else % insert nans between patches
    xs(end+1,:) = nan;
    us(end+1,:,:) = nan;
    ustd(end+1,:,:) = nan;
end
us=reshape(permute(us,[1 3 2]),[],length(ts));
ustd=reshape(permute(ustd,[1 3 2]),[],length(ts));
```

Now plot two space-time graphs. The first is every time step over a meso-time to see the oscillation and decay of the fast sub-patch diffusions. The second is subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale diffusion over the heterogeneous lattice.

```
for p=1:2
  switch p
  case 1, j=find(ts<0.01);
```

```
  case 2, [~,j]=min(abs(ts(:)-linspace(ts(1),ts(end),50)));
  end
  figure(p),clf
  mesh(ts(j),xs(:),us(:,j))
  view(60,40), colormap(0.8*hsv)
  xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
  ifOurCf2eps([mfilename 'U' num2str(p)])
end
pause(3)
```

Compute Jacobian and its spectrum Let's explore the Jacobian dynamics for a range of orders of interpolation, all for the same patch design and heterogeneity. Here use a smaller ratio, and more patches, as we do not plot.

```
nPatch = 13
ratio = 0.01;

leadingEvals=[];
for ord=0:2:8
  ordInterp=ord
  configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
      ,ord,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr);
```

Form the Jacobian matrix, linear operator, by numerical construction about a zero field. Use `i` to store the indices of the micro-grid points that are interior to the patches and hence are the system variables.

```
  u0 = zeros(nSubP,1,nEnsem,nPatch);
  u0([1 end],:,:,:)=nan; u0=u0(:);
  i=find(~isnan(u0));
  nJ=length(i);
  Jac=nan(nJ);
  for j=1:nJ
    u0(i)=((1:nJ)==j);
    dudt=patchSys1(0,u0);
    Jac(:,j)=dudt(i);
  end
  nonSymmetric=norm(Jac-Jac')
  assert(nonSymmetric<5e-9,'failed symmetry')
  Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in Table 3.1: the spectral interpolation is effectively exact for the macroscale; quadratic interpolation is usually quantitatively in error; quartic interpolation appears to be the lowest order for reliable quantitative accuracy.

The number of zero eigenvalues, `nZeroEv`, indicates the number of decoupled systems in this patch configuration.

Table 3.1: example parameters and list of eigenvalues (every fourth one listed is sufficient due to symmetry): `nPatch` = 19, `ratio` = 0.1, `nSubP` = 5. The columns are for various `ordCC`, in order: 0, spectral interpolation; 2, quadratic; 4, quartic; and 6, sixth order.

```
cHetr =
       6.9617
       0.4217
       2.0624
leadingEvals =
        2e-11       -2e-12        4e-12       -2e-11
      -0.9999      -1.5195      -1.0127      -1.0003
      -3.9992      -11.861      -4.7785      -4.0738
      -8.9960      -45.239      -17.164      -10.703
      -15.987      -116.27      -56.220      -30.402
      -24.969      -230.63      -151.74      -92.830
      -35.936      -378.80      -327.36      -247.37
      -48.882      -535.89      -570.87      -521.89
      -63.799      -668.21      -818.33      -855.72
      -80.678      -743.96      -976.57      -1093.4
       -29129       -29233       -29227       -29222
       -29151       -29234       -29229       -29223
```

```
280   [evecs,evals]=eig(Jac);
281   eval=-sort(-diag(real(evals)));
282   nZeroEv=sum(eval(:)>-1e-5)
283   leadingEvals=[leadingEvals eval(1:3*nPatch)];
284 %  leadingEvals=[leadingEvals eval([1, (nZeroEv+1):2:(nZeroEv*nPatch+4)])];
```

End of the for-loop over orders of interpolation, and output the tables of eigenvalues.

```
291 end
292 disp('     spectral    quadratic      quartic  sixth-order  ...')
293 leadingEvals=leadingEvals
```

End of the main script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

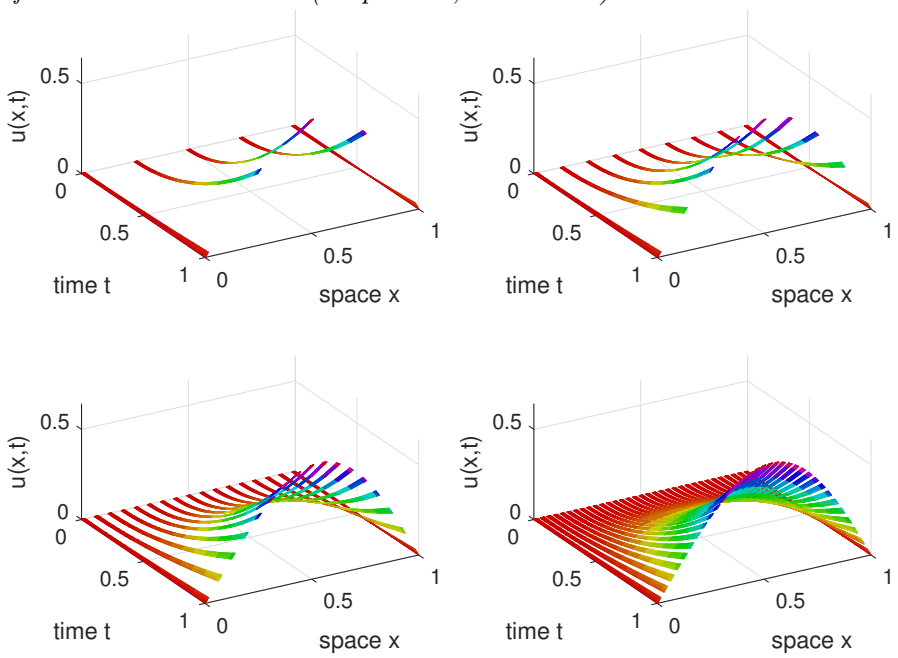
This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSys1`, Section 3.2), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector of diffusivities c_i, and possibly Burgers' advection coefficients b_i, have previously been stored in struct `patches.cs`.

```
21 function ut = heteroDiff(t,u,patches)
22   dx = diff(patches.x(2:3));   % space step
23   i = 2:size(u,1)-1;   % interior points in a patch
```

```
24    ut = nan+u;          % preallocate output array
25    ut(i,:,:,:) = diff(patches.cs(:,1,:).*diff(u))/dx^2;
26    % possibly include heterogeneous Burgers' advection
27    if size(patches.cs,2)>1 % check for advection coeffs
28        buu = patches.cs(:,2,:).*u.^2;
29        ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
30    end
31  end% function
```

Fin.

Figure 3.8: diffusion field $u(x,t)$ of the patch scheme applied to the forced heterogeneous diffusive (3.3). Simulate for $5, 9, 17, 33$ patches and compare to the full-domain simulation (65 patches, not shown).



3.6 `Eckhardt2210eg2`: example of a 1D heterogeneous diffusion by simulation on small patches

Plot an example simulation in time generated by the patch scheme applied to macroscale forced diffusion through a medium with microscale heterogeneity in space. This is more-or-less the second example of Eckhardt & Verfürth (2022) [§6.2.1].

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice forced diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[a_{i-1/2}\delta u_i] + f_i(t), \tag{3.3}$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which has some given known periodicity ϵ.

Here use period $\epsilon = 1/130$ (so that computation completes in seconds). The patch scheme computes only on a fraction of the spatial domain, see Figure 3.8. Compute *errors* as the maximum difference (at time $t = 1$) between the patch scheme prediction and a full-domain simulation of the same underlying spatial discretisation (which here has space step 0.00128).

patch spacing H	0.25	0.12	0.06	0.03
exp-sine-forcing error	8E−3	2E−3	3E−4	2E−5
parabolic-forcing error	9E−9	4E−9	1E−9	0.06E−9

The smooth sine-forcing leads to errors that appear due to patch scheme

and its interpolation. The parabolic-forcing errors appear to be due to the integration errors of `ode15s` and not at all due to the patch scheme. In comparison, Eckhardt & Verfürth (2022) reported much larger errors in the range 0.001–0.1 (Figure 3).

3.6.1 Simulate heterogeneous diffusion systems

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and coefficients to match Eckhardt2210.04536 §6.2.1. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then the heterogeneity is repeated to fill each patch.

```
clear all
%global OurCf2eps, OurCf2eps=true %option to save plots
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = 1./(2-cos(2*pi*y(1:mPeriod)))
global microTimePeriod; microTimePeriod=0;
```

Set the spatial period ϵ, via integer $1/\epsilon$, and other parameters.

```
maxLog2Nx = 6
nPeriodsPatch = 2 % any integer
rEpsilon = nPeriodsPatch*(2^maxLog2Nx+1) % up to 200 say
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
tol=1e-9;
```

Loop to explore errors on various sized patches.

```
Us=[]; DXs=[]; % for storing results to compare
iPP=0; I=nan;
for log2Nx = 2:maxLog2Nx
    nP = 2^log2Nx+1
```

Determine indices of patches that are common in various resolutions

```
    if isnan(I), I=1:nP; else I=2*I-1; end
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.3) solved on domain $[0, 1]$, with `nP` patches, and say fourth order interpolation to provide the edge-values. Setting `patches.EdgyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```
    global patches
    ordCC = 4
    configPatches1(@heteroDiffF,[0 1],'equispace',nP ...
        ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
    DX = mean(diff(squeeze(patches.x(1,1,1,:))))
    DXs=[DXs;DX];
```

Set the forcing coefficients, either the original parabolic, or exp-sinusoidal.

```
140     if 0 % given forcing is exact
141       patches.f1=2*( patches.x-patches.x.^2 );
142       patches.f2=2*0.5+0*patches.x;
143     else% simple exp.sine forcing
144       patches.f1=sin(pi*patches.x).*exp(patches.x);
145       patches.f2=pi/2*sin(pi*patches.x).*exp(patches.x);
146     end%if
```

Simulate Set the initial conditions of a simulation to be zero. Integrate to time 1 using standard integrators.

```
157     u0 = 0*patches.x;
158     tic
159     [ts,us] = ode15s(@patchSys1, [0 1], u0(:));
160     cpuTime=toc
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so adjoin a row of `nan`s in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nan`s, and reshape again.

```
173     xs = squeeze(patches.x);
174     us = patchEdgeInt1( permute( reshape(us ...
175         ,length(ts),nSubP,1,nP) ,[2 1 3 4]) );
176     us = squeeze(us);
177     xs(end+1,:) = nan;  us(end+1,:,:) = nan;
178     uss=reshape(permute(us,[1 3 2]),[],length(ts));
```

Plot a space-time surface of field values over the macroscale duration of the simulation.

```
186     iPP=iPP+1;
187     if iPP<=4 % only draw four subplots
188       figure(1), if iPP==1, clf(), end
189       subplot(2,2,iPP)
190       mesh(ts,xs(:),uss)
191       if iPP==1, uMax=ceil(max(uss(:))*100)/100, end
192       view(60,40), colormap(0.8*hsv), zlim([0 uMax])
193       xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
194       drawnow
195     end%if
```

At the end of the `log2Nx`-loop, store field at the end-time from centre region of each patch for comparison.

```
203     i=nPeriodsPatch/2*mPeriod+1+(-mPeriod/2+1:mPeriod/2);
204     Us(:,:,iPP)=squeeze(us(i,end,I));
205     Xs=squeeze(patches.x(i,1,1,I));
206     if iPP>1
207        assert(norm(Xs-Xsp)<tol,'sampling error in space')
208        end
```

```
209     Xsp=Xs;
210 end%for log2Nx
211 ifOurCf2eps(mfilename) %optionally save plot
```

Assess errors by comparing to the full-domain solution

```
217 DXs=DXs
218 Uerr=squeeze(max(max(abs(Us-Us(:,:,end)))))
219 figure(2),clf,
220 loglog(DXs,Uerr,'o:')
221 xlabel('$H$'),ylabel('error')
222 ifOurCf2eps([mfilename 'Errs']) %optionally save plot
```

3.6.2 `heteroDiffF()`: forced heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches with forcing and with microscale boundary conditions on the macroscale boundaries. Computes the time derivative at each point in the interior of a patch, output in `ut`. The column vector of diffusivities a_i has been stored in struct `patches.cs`, as has the array of forcing coefficients.

```
17 function ut = heteroDiffF(t,u,patches)
```

Cater for the two cases: one of a non-autonomous forcing oscillating in time when `microTimePeriod` > 0, or otherwise the case of an autonomous diffusion constant in time.

```
26   global microTimePeriod
27   if microTimePeriod>0 % optional time fluctuations
28      at = cos(2*pi*t/microTimePeriod)/30;
29   else at=0; end
```

Two basic parameters, and initialise result array to NaNs.

```
35   dx = diff(patches.x(2:3));   % space step
36   i = 2:size(u,1)-1;   % interior points in a patch
37   ut = nan+u;          % preallocate output array
```

The macroscale Dirichlet boundary conditions are zero at the extreme edges of the two extreme patches.

```
44   u( 1 ,:,:, 1 )=0; % left-edge of leftmost is zero
45   u(end,:,:,end)=0; % right-edge of rightmost is zero
```

Code the microscale forced diffusion.

```
51   ut(i,:,:,:) = diff((patches.cs(:,1,:)+at).*diff(u))/dx^2 ...
52       +patches.f2(i,:,:,:)*t^2+patches.f1(i,:,:,:)*t;
53 end% function
```

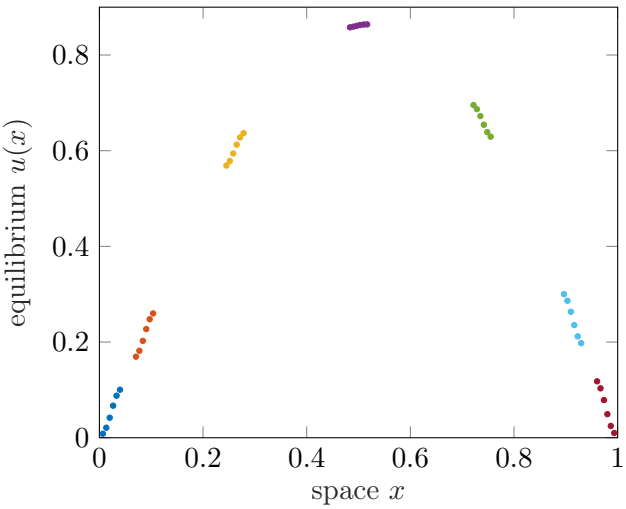


Figure 3.9: Equilibrium of the heterogeneous diffusion problem with forcing the same as that applied at time $t = 1$, and for relatively large $\epsilon = 0.04$ so we can see the patches. By default this code sets $\epsilon = 0.004$ whence the microscale heterogeneity and patches are tiny.

3.7 `EckhardtEquilib`: find an equilibrium of a 1D heterogeneous diffusion via small patches

Sections 3.6 and 3.6.2 describe details of the problem and more details of the following configuration. The aim is to find the equilibrium, Figure 3.9, of the forced heterogeneous system with a forcing corresponding to that applied at time $t = 1$. Computational efficiency comes from only computing the microscale heterogeneity on small spatially sparse patches, potentially much smaller than those shown in Figure 3.9.

First configure the patch system Establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and coefficients to match Eckhardt & Verfürth (2022) [§6.2.1].

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plots
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = 1./(2-cos(2*pi*y(1:mPeriod)))
global microTimePeriod; microTimePeriod=0;
```

Set the number of patches, the number of periods per patch, and the spatial period ϵ, via integer $1/\epsilon$.

```
nPatch = 7
nPeriodsPatch = 1 % any integer
rEpsilon = 25 % 25 for graphic, up to 2000 say
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.3) solved on domain $[0, 1]$, with Chebyshev-like distribution of patches, and say fourth order interpolation to provide the

edge-values. Use ‘edgy’ interpolation.

```
77 ordCC = 4
78 configPatches1(@heteroDiffF,[0 1],'chebyshev',nPatch ...
79     ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
```

Set the forcing coefficients, either the original parabolic, or exp-sinusoidal. At time $t = 1$ the resultant forcing we actually apply here is simply the sum of the two components.

```
88 if 0 % given forcing
89   patches.f1 = 2*( patches.x-patches.x.^2 );
90   patches.f2 = 2*0.5+0*patches.x;
91 else% simple exp-sine forcing
92   patches.f1 = sin(pi*patches.x).*exp(patches.x);
93   patches.f2 = pi/2*sin(pi*patches.x).*exp(patches.x);
94 end%if
```

Find equilibrium with fsolve We seek the equilibrium for the forcing that applies at time $t = 1$ (as if that specific forcing were applying for all time). For this linear problem, it is computationally quicker using a linear solver, but `fsolve` is quicker in human time, Start the search from a zero field.

```
107 u = 0*patches.x;
```

But set patch-edge values to `Nan` in order to use `patches.i` to index the interior sub-patch points as they are the variables.

```
115 u([1 end],:,:,:) = nan;
116 patches.i = find(~isnan(u));
```

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 3.15).

```
124 [u(patches.i),res] = fsolve(@theRes,u(patches.i));
125 normRes = norm(res)
```

Plot the equilibrium see Figure 3.9.

```
132 clf, plot(squeeze(patches.x),squeeze(u),'.')
133 xlabel('space $x$'),ylabel('equilibrium $u(x)$')
134 ifOurCf2tex(mfilename)%optionally save
```

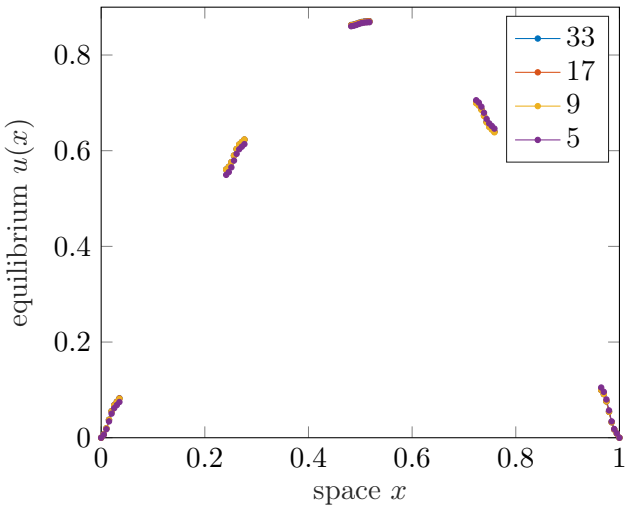


Figure 3.10: Equilibrium of the heterogeneous diffusion problem for relatively large $\epsilon = 0.03$ so we can see the patches. The solution is obtained with various numbers of patches, but we only compare solutions in these five common patches.

3.8 `EckhardtEquilibErrs`: explore errors in equilibria of a 1D heterogeneous diffusion on small patches

Section 3.7 finds the equilibrium, of the forced heterogeneous system with a forcing corresponding to that applied at time $t = 1$. Computational efficiency comes from only computing the microscale heterogeneity on small spatially sparse patches. Here we explore the errors as the number N of patches increases, see Figures 3.10 and 3.11. Find mean-abs errors to be the following:

	$N =$	5	9	17	33	65
equispace	second-order	8E−3	1E−2	1E−2	4E−3	9E−4
equispace	fourth-order	2E−3	7E−4	1E−4	9E−6	5E−7
equispace	sixth-order	2E−3	2E−5	4E−7	1E−8	2E−10
chebyshev	second-order	4E−2	6E−2	3E−2	2E−2	2E−2
chebyshev	fourth-order	9E−4	3E−3	6E−4	3E−4	2E−4
chebyshev	sixth-order	9E−4	3E−5	1E−5	4E−6	1E−6
usergiven	second-order	4E−2	6E−2	3E−2	9E−3	2E−3
usergiven	fourth-order	8E−4	3E−3	6E−4	4E−5	2E−6
usergiven	sixth-order	8E−4	3E−5	1E−5	2E−7	3E−9

For 'chebyshev' this assessment of errors is a bit dodgy as it is based only on the centre and boundary patches. The 'usergiven' distribution is for overlapping patches with Chebyshev distribution of centres—a spatial 'christmas tree'[4]. Curiously, and with above caveats, here my 'smart' chebyshev is the worst, the overlapping Chebyshev is good, but *equispace appears usually the best.*

The above errors are for simple sin forcing. What if we make not so simple with exp modification of the forcing? The errors shown below are very little

[4] But the error assessment is with respect to finest patch-grid, no longer with a full domain solution

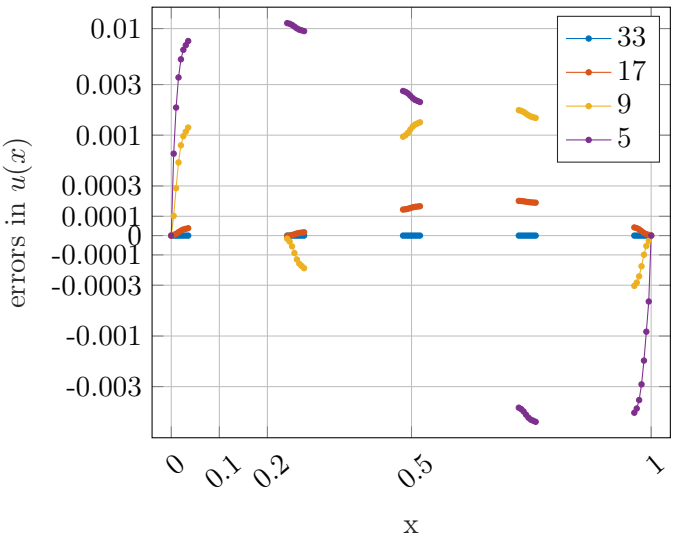


Figure 3.11: Errors in the equilibrium of the heterogeneous diffusion problem for relatively large $\epsilon = 0.03$. The solution is obtained with various numbers of patches, but we only plot the errors within these five common patches.

different (despite the magnitude of the solution being a little larger).

	$N =$	5	9	17	33	65
equispace	fourth-order	4E−3	7E−4	1E−4	8E−6	5E−7
chebyshev	fourth-order	7E−4	2E−3	5E−4	3E−4	1E−4
usergiven	fourth-order	2E−3	3E−3	5E−4	4E−5	2E−6

Clear, and initiate global patches. Choose the type of patch distribution to be either 'equispace', 'chebyshev', or 'usergiven'. Also set order of interpolation (fourth-order is good start).

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plots
switch 1
    case 1, Dom.type = 'equispace'
    case 2, Dom.type = 'chebyshev'
    case 3, Dom.type = 'usergiven'
    end% switch
ordInt = 4
```

First configure the patch system Establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and coefficients to match Eckhardt2210.04536 §6.2.1.

```
mPeriod = 6
z = (0.5:mPeriod)'/mPeriod;
a = 1./(2-cos(2*pi*z))
global microTimePeriod; microTimePeriod=0;
```

To use a hierarchy of patches with `nPatch` of 5, 9,17, ..., we need up to N patches plus one `dx` to fit into the domain interval. Cater for up to some full-domain simulation—can compute `log2Nmax` = 129 ($\epsilon = 0.008$) in a few seconds:

```
log2Nmax = 7 % 5 for plots, 7 for choice
nPatchMax=2^log2Nmax+1
```

Set the periodicity ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
epsilon = 1/(nPatchMax*nPeriodsPatch+1/mPeriod)
dx = epsilon/mPeriod
```

For various numbers of patches Choose five to be the coarsest number of patches. Want place to store common results for the solutions. Assign `Ps` to be the indices of the common patches: for equispace set to the five common patches, but for chebyshev the only common ones are the three centre and boundary-adjacent patches.

```
us=[]; xs=[]; nPs=[];
for log2N=log2Nmax:-1:2
    if log2N==log2Nmax
         Ps=linspace(1,nPatchMax ...
           ,5-2*all(Dom.type=='chebyshev'))
    else Ps=(Ps+1)/2
    end
```

Set the number of patches in $(0, 1)$:

```
    nPatch = 2^log2N+1
```

In the case of 'usergiven', we choose standard Chebyshev distribution of the centre of the patches, which involves overlapping of patches near the boundaries! (instead of the coded chebyshev which has a boundary layer of non-overlapping patches and a Chebyshev within the interior).

```
    if all(Dom.type=='usergiven')
      halfWidth=dx*(nSubP-1)/2;
      X1 = 0+halfWidth; X2 = 1-halfWidth;
      Dom.X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
    end
```

Configure the patches:

```
    configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
        ,ordInt,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
```

Set the forcing coefficients, either the original parabolic, or sinusoidal. At time $t = 1$ the resultant forcing we actually apply here is simply the sum of the two components.

```
    if 0 %given forcing gives exact answers for ordInt=4 !
      patches.f1 = 2*( patches.x-patches.x.^2 );
      patches.f2 = 2*0.5+0*patches.x;
    else% simple exp-sine forcing
      patches.f1 = sin(pi*patches.x).*exp(patches.x);
```

```
241       patches.f2 = pi/2*sin(pi*patches.x).*exp(patches.x);
242     end%if
```

Solve for steady state Set initial guess of either zero or a subsample of the next finer solution, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
254     if log2N==log2Nmax
255       u0 = zeros(nSubP,1,1,nPatch);
256     else u0 = u0(:,:,:,1:2:end);
257     end
258     u0([1 end],:) = nan;
259     patches.i = find(~isnan(u0));
260     nVariables = numel(patches.i)
```

Solve via `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 3.15).

```
269     tic;
270     [uSoln,resSoln] = fsolve(@theRes,u0(patches.i) ...
271         ,optimoptions('fsolve','Display','off'));
272     fsolveTime = toc
```

Store the solution into the patches, and give magnitudes—Inf norm is max(abs()).

```
279     normSoln = norm(uSoln,Inf)
280     normResidual = norm(resSoln,Inf)
281     u0(patches.i) = uSoln;
282     u0 = patchEdgeInt1(u0);
283     u0( 1 ,:,:, 1 ) = 0;
284     u0(end,:,:,end) = 0;
```

Concatenate the solution on common patches into stores.

```
290     us=cat(3,us,squeeze(u0(:,:,:,Ps)));
291     xs=cat(3,xs,squeeze(patches.x(:,:,:,Ps)));
292     nPs = [nPs;nPatch];
```

End loop. Check grids were aligned, then compute errors compared to the full-domain solution.

```
300 end%for log2N
301 assert(max(abs(reshape(diff(xs,1,3),[],1)))<1e-12,'x-coord failure')
302 errs = us-us(:,:,1);
303 meanAbsErrs = mean(abs(reshape(errs,[],size(us,3))))
304 ratioErrs = meanAbsErrs(2:end)./meanAbsErrs(1:end-1)
```

Plot solution in common patches First adjoin NaNs to separate patches, and reshape.

```
x=xs(:,:,1); u=us;
x(end+1,:)=nan; u(end+1,:)=nan;
u=reshape(u,numel(x),[]);
```

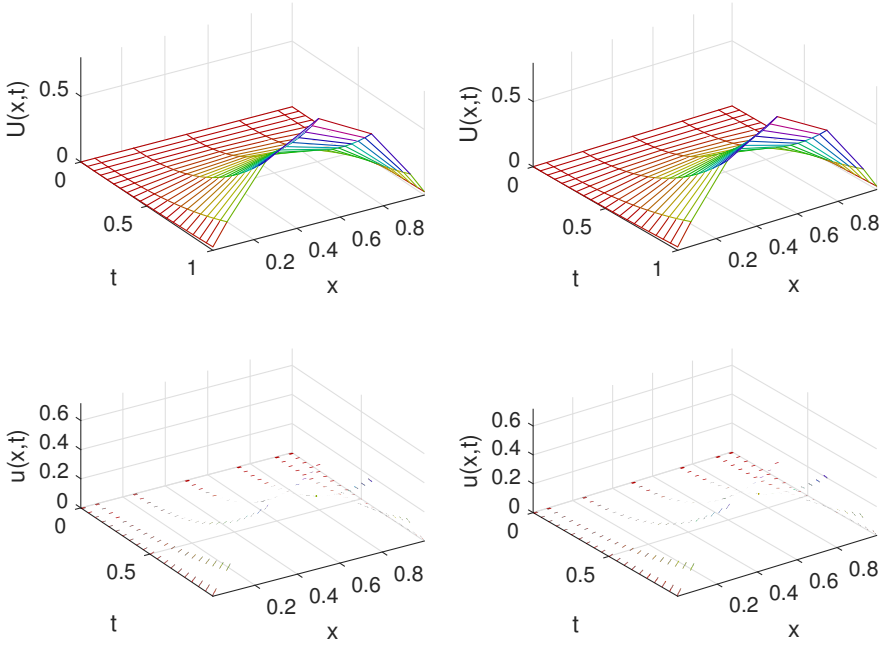
Reshape solution field.

```
figure(1),clf
plot(x(:),u,'.-'), legend(num2str(nPs))
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([mfilename 'us'])%optionally save
```

Plot errors Use quasi-log axis to separate the errors.

```
err = u(:,1)-u;
figure(2), clf
h=plot(x(:),err,'.-'); legend(num2str(nPs))
quasiLogAxes(h,10,sqrt(prod(meanAbsErrs(2:3))))
xlabel('space $x$'), ylabel('errors in $u(x)$')
ifOurCf2tex(mfilename)%optionally save
```

Figure 3.12: diffusion field $u(x,t)$ of the patch scheme applied to the forced space-time heterogeneous diffusive (3.4). Simulate for seven patches (with a 'Chebyshev' distribution): the top stereo pair is a mesh plot of a macroscale value at the centre of each spatial patch at each projective integration time-step; the bottom stereo pair shows the corresponding tiny space-time patches in which microscale computations were carried out.



3.9 `Eckhardt2210eg1`: example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches

An example simulation in time generated by projective integration allied with the patch scheme applied to forced diffusion in a medium with microscale heterogeneity in both space and time. This is more-or-less the first example of Eckhardt & Verfürth (2022) [§6.2].

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $u_i(t)$, simulate the microscale lattice forced diffusion system

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2}\delta[a_{i-1/2}(t)\delta u_i] + f_i(t), \tag{3.4}$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which has given periodicity ϵ in space, and periodicity ϵ^2 in time. Figure 3.12 shows an example patch simulation.

The approximate homogenised PDE is $U_t = A_0 U_{xx} + F$ with $U = 0$ at $x = 0, 1$. Its slowest mode is then $U = \sin(\pi x)e^{-A_0\pi^2 t}$. When $A_0 = 3.3524$ as in Eckhardt then the rate of evolution is about 33 which is relatively fast on the simulation time-scale of $T = 1$. Let's slow down the dynamics by reducing diffusivities by a factor of 30, so effectively $A_0 \approx 0.1$ and $A_0\pi^2 \approx 1$.

Also, in the microscale fluctuations change the time variation to cosine, not

its square (because I cannot see the point of squaring it!).

The highest wavenumber mode on the macro-grid of patches, spacing H, is the zig-zag mode on $\dot{U}_i = A_0(U_{I+1} - 2U_I + U_{I-1})/H^2 + F_I$ which evolves like $U_I = (-1)^I e^{-\alpha t}$ for the fastest 'slow rate' of $\alpha = 4A_0^2/H^2$. When $H = 0.2$ and $A_0 \approx 0.1$ this rate is $\alpha \approx 10$.

Here use period $\epsilon = 1/100$ (so that computation completes in seconds, and because we have slowed the dynamics by 30). The patch scheme computes only on a fraction of the spatial domain. Projective integration computes only on a fraction of the time domain determined by the 'burst length'.

3.9.1 Simulate heterogeneous diffusion systems

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice, and coefficients inspired by Eckhardt2210.04536 §6.2. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then the heterogeneity is repeated to fill each patch. If an odd number of odd-periods in a patch, then the centre patch is a grid point of the field u, otherwise the centre patch is at a half-grid point.

```
clear all
%global OurCf2eps, OurCf2eps=true %option to save plots
mPeriod = 6
y = linspace(0,1,mPeriod+1)';
a = ( 3+cos(2*pi*y(1:mPeriod)) )/30
A0 = 1/mean(1./a) % roughly the effective diffusivity
```

The microscale diffusivity has an additional additive component of $+\frac{1}{30}\cos(2\pi t/\epsilon^2)$ which is coded into time derivative routine via global `microTimePeriod`.

Set the periodicity, via integer $1/\epsilon$, and other parameters.

```
nPeriodsPatch = 2 % any integer
rEpsilon = 100
dx = 1/(mPeriod*rEpsilon+1)
nSubP = nPeriodsPatch*mPeriod+2
tol=1e-9;
```

Set the time periodicity (global).

```
global microTimePeriod
microTimePeriod = 1/rEpsilon^2
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.4) solved on macroscale domain $[0, 1]$, with `nPatch` patches, and say fourth-order interpolation to provide the edge-values of the inter-patch coupling conditions. Distribute the patches either equispaced or chebyshev. Setting `patches.EdgyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```
nPatch = 7
ordCC = 4
```

```
Dom = 'chebyshev'
global patches
configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
    ,ordCC,dx,nSubP,'EdgyInt',true,'hetCoeffs',a);
DX = mean(diff(squeeze(patches.x(1,1,1,:))))
```

Set the forcing coefficients as the odd-periodic extensions, accounting for roundoff error in `f2`.

```
if 0 % given forcing
  patches.f1=2*( patches.x-patches.x.^2 );
  patches.f2=2*0.5+0*patches.x;
else% simple sine forcing
  patches.f1=sin(pi*patches.x);
  patches.f2=pi/2*sin(pi*patches.x);
end%if
```

Simulate Set the initial conditions of a simulation to be zero. Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
u0 = 0*patches.x;
u0([1 end],:) = nan;
```

Set the desired macro- and microscale time-steps over the time domain. The macroscale step is in proportion to the effective mean diffusion time on the macroscale, here $1/(A_0\pi^2) \approx 1$ so for macro-scale error less than 1% need $\Delta t < 0.24$, so use 0.1 say.

The burst time depends upon the sub-patch effective diffusion rate β where here rate $\beta \approx \pi^2 A_0/h^2 \approx 2000$ for patch width $h \approx 0.02$: use the formula from the Manual, with some extra factor, and rounded to the nearest multiple of the time micro-periodicity.

```
ts = linspace(0,1,21)
h=(nSubP-1)*dx;
beta = pi^2*A0/h^2 % slowest rate of fast modes
burstT = 2.5*log(beta*diff(ts(1:2)))/beta
burstT = max(10,round(burstT/microTimePeriod))*microTimePeriod +1e-12
addpath('../../ProjInt')
```

Time the projective integration simulation.

```
tic
[us,tss,uss] = PIRK2(@heteroBurstF, ts, u0(:), burstT);
cputime=toc
```

Plot space-time surface of the simulation First, just a macroscale mesh plot—stereo pair.

```
xs=squeeze(patches.x);
Xs=mean(xs);
Us=squeeze(mean( reshape(us,length(ts),[],nPatch), 2,'omitnan'));
```

```
figure(1),clf
for k = 1:2, subplot(2,2,k)
  mesh(ts,Xs(:),Us')
  ylabel('space $x$'), xlabel('time $t$'), zlabel('$U(x,t)$')
  colormap(0.8*hsv), axis tight, view(62-4*k,45)
end
```

Second, plot a surface detailing the microscale bursts—stereo pair. Do not bother with the patch-edge values. Optionally save to Figs folder.

```
xs([1 end],:) = nan;
for k = 1:2, subplot(2,2,2+k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('space $x$'), xlabel('time $t$'), zlabel('$u(x,t)$')
  colormap(0.7*hsv), axis tight, view(62-4*k,45)
end
ifOurCf2eps(mfilename)
```

3.9.2 `heteroBurstF()`: a burst of heterogeneous diffusion

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSys1`. Try `ode23`, although `ode45` may give smoother results. Sample every period of the microscale time fluctuations (or, at least, close to the period).

```
function [ts, ucts] = heteroBurstF(ti, ui, bT)
  global microTimePeriod
  [ts,ucts] = ode45( @patchSys1,ti+(0:microTimePeriod:bT),ui(:));
end
```

3.10 `homoLanLif1D`: computational homogenisation of a 1D heterogeneous Landau–Lifshitz by simulation on small patches

The Landau–Lifshitz equation describes the precessional motion of magnetization $\vec{M}$ in a solid (see *Landau–Lifshitz–Gilbert equation* in Wikipedia). In a medium with microscale heterogeneity $a(x)$, and with phenomenological damping parameter α, we explore the dynamics of $\vec{M}(x,t)$ governed by the nonlinear Landau–Lifshitz PDE (Leitenmaier & Runborg 2021, (1.1)) [5]

$$\vec{M}_t = -\vec{M} \times \vec{H} - \alpha \vec{M} \times (\vec{M} \times \vec{H}), \quad \vec{H} := \vec{\nabla} \cdot (a \vec{\nabla} \vec{M}).$$

Note, for every x, $|\vec{M}(x,t)|$ is constant in time due to $\vec{M} \cdot \vec{M}_t = 0$ for every x, t. We normally set $|\vec{M}(x,0)| = 1$.

Figure 3.13 shows an example simulation in time generated by the patch scheme applied to the above Landau–Lifshitz PDE on the spatial domain $[0,1]$ with domain boundary conditions of 1-periodicity. The inter-patch coupling is realised by interpolation of the patch's next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems (quartic interpolation appears to be the lowest order that generally gives good accuracy). With damping parameter $\alpha = 0.001$ then the largest few macroscale modes decay with rate roughly 0.1, and so are negligibly damped over a time of 0.1.

Suppose the spatial microscale lattice is at points x_i, with constant spacing dx. With dependent variables $\vec{M}_i(t)$, simulate the microscale lattice system

$$\vec{M}_{i\,t} = -\vec{M}_i \times \vec{H}_i - \alpha \vec{M}_i \times (\vec{M}_i \times \vec{H}_i), \quad \vec{H}_i := \frac{1}{dx^2} \delta [a_{i-1/2} \delta \vec{M}_i],$$

in terms of the centred difference operator δ. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which we assume to have some given known periodicity (Leitenmaier & Runborg 2021, pp.6,27). Figure 3.13 shows a patch simulation of this system: observe the effects of the heterogeneity within each patch.

Parameters There are two closely related examples (Leitenmaier & Runborg 2021, pp.6,27), that we distinguish here with parameter `ex5p1`: set to either zero or one. The Landau–Lifshitz dissipation parameter α should be small. If the initial conditions are smooth, then `ode15s` has no problems for $\alpha = 0.001$. [6]

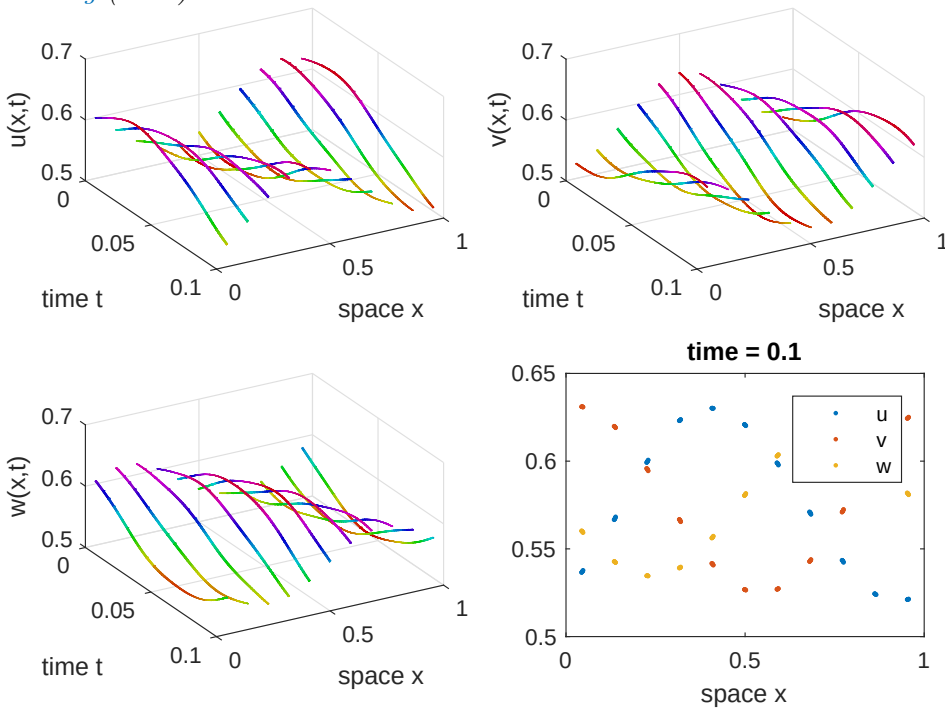
```
global alpha ex5p1
ex5p1 = 0; % set to 1 for L&O example of p.27
alpha = 0.001 % phenomenological damping parameter
```

The physical microscale periodicity of the heterogeneity is ϵ (ϵ is *not* the patch scale ratio):

[5] Recall $a \times (b \times c) = (a \cdot c)b - (a \cdot b)c$

[6] But, add randomness to the initial conditions and the computation appears unstable with `ode15s` when $\alpha < 0.2$. However, `ode23` may be stable? for $\alpha = 0.01$ albeit expensively taking 10^7 time-steps per second (due to microscale oscillations of frequency up to 10^5–10^6).

Figure 3.13: magnetic field $\vec{M}(x,t) = (u,v,w)$ of the gap-tooth scheme applied to the heterogeneous Landau–Lifshitz PDE *to show the emergent macroscale wave-like dynamics. This simulation uses eleven patches in space of size ratio* 0.055*. Compare the time $t = 0.1$ graph with Fig. 2.1 of Leitenmaier & Runborg (2021).*




```
epsilon = 1/200/(1+ex5p1) %pp.6,27
```

3.10.1 Script code to simulate heterogeneous diffusion systems

This example script implements the following patch/gap-tooth scheme.

1. configPatches1
2. ode15s $\mapsto$ patchSys1 $\mapsto$ heteroLanLif1D
3. plot the simulation

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice with values of the column vector from Leitenmaier & Runborg (2021) [pp.6,27]. Later, the heterogeneity is repeated to fill each patch.

```
dx = 1/2000 %1/6000 %p.27
mPeriod = round(epsilon/dx)
a = 1 + 0.5*sin(2*pi*(0.5:mPeriod)'/mPeriod); %p.6
```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.2) solved on 1-periodic domain, with maybe 24 patches, but 11 is enough, here each patch of size ratio to fit one period of the heterogeneity in each patch, and spectral inter-patch interpolation to provide the patch edge-values. Invoking `EdgyInt` means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch

values).

```
global patches
nPatch = 11 %24 %p.6, odd is slightly cleaner
nSubP = mPeriod+2
ratio = nPatch*epsilon
configPatches1(@heteroLanLif1D,[0 1],nan,nPatch ...
    ,0,ratio,nSubP,'EdgyInt',true ...
    ,'hetCoeffs',a);
assert(abs(dx-diff(patches.x(2:3)))<1e-10 ...
      ,'microscale grid spacing error')
```

Simulate Set the initial conditions of a simulation to be that of Leitenmaier & Runborg (2021) [pp.6], except possibly perturbed by random microscale noise. Scale the initial conditions so that $|\vec{M}(x,0)| = 1$.

```
u0 = 0.5+exp(-0.1*cos(2*pi*(patches.x-0.32)));
v0 = 0.5+exp(-0.2*cos(2*pi*patches.x)) +0*randn(size(patches.x));
w0 = 0.5+exp(-0.1*cos(2*pi*(patches.x-0.75)));
M0 = [ u0 v0 w0 ]./sqrt(u0.^2+v0.^2+w0.^2);
dM0dt = patchSys1(0,M0(:));
```

Integrate using standard integrators.

```
tic
[ts,Ms] = ode15s(@patchSys1, [0 0.1], M0(:));
cpuTime=toc
sizeMs=size(Ms)
```

Reshape results for processing. For simplicity, set edge values to `nan`s. For the field values (which are rows in `Ms`) we need to reshape, permute, and reshape again.

```
xs = squeeze(patches.x);
Ms = reshape(Ms,length(ts),nSubP,3,nPatch);
Ms(:,[1 end],:,:) = nan; % nan patch edges
Ms = reshape( permute(Ms,[2 4 1 3]) ,[],length(ts),3);
```

Check on constancy of $|\vec{M}(x,t)|$ in time. The mean and standard deviation appears to show that, with `ode15s`, they are constant to errors typically 10^{-5}.

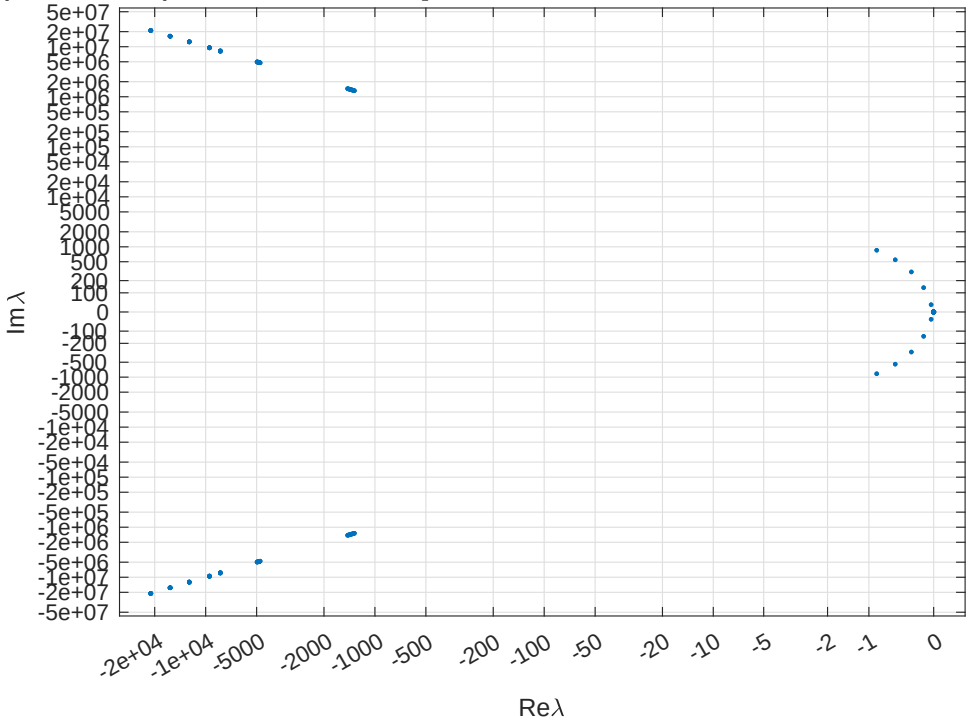
```
Mabs = sqrt( sum(Ms.^2,3) );
meanMabs = mean(Mabs(:),'omitnan')
stdevMabs = std(Mabs(:),'omitnan')
```

Plot space-time surface of the simulation Choose whether to save some plots, or not.

```
global OurCf2eps
OurCf2eps = false;
```

Subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale modes over the heterogeneous lattice.

Figure 3.14: spectrum of eigenvalues of the multiscale patch scheme applied to the Landau–Lifshitz PDE. *The macroscale eigenvalues are clearly separated from those of the microscale sub-patch modes.*




```
figure(1),clf
if length(ts)>50
  [~,j]=min(abs(ts(:)-linspace(ts(1),ts(end),50)));
  else j=1:length(ts); end
uvw='uvw';
for p=1:3
  subplot(2,2,p)
  mesh(ts(j),xs(:),Ms(:,j,p))
  view(60,40), colormap(0.8*hsv)
  xlabel('time $t$'), ylabel('space $x$')
  zlabel(['$' uvw(p) '(x,t)$'])
end
```

Final time plot to compare with Fig. 2.1 of Leitenmaier & Runborg (2021).

```
subplot(2,2,4)
plot(xs(:),squeeze(Ms(:,end,:)),'.')
xlabel('space $x$'), legend(uvw(1),uvw(2),uvw(3))
title(['time = ' num2str(ts(end),4)])
ifOurCf2eps([mfilename 'uvw'])
```

3.10.2 Spectrum of the coded patch system

It appears the spectrum has the following properties as shown by Figure 3.14, with $N = \texttt{nPatch}$ and $n = \texttt{nSubP} - 2$, and on base of $\vec{M} = \vec{1}/\sqrt{3}$.

- A (near) zero eigenvalue for each and every microscale lattice point (nN) due to $|\vec{M}(x,t)|$ being constant in time, for every x. Presumably near zero (roughly 10^{-2}) due to round-off error.
- $2N$ macroscale eigenvalues, including a pair of (near) zero eigenvalues of macroscale conservation, and others ranging from $27(-\alpha \pm i)$ to $(-6.4\alpha \pm 7.4i)(N-1)^2$.
- $2(n-1)N$ fast eigenvalues, more negative than about $-\alpha \cdot 10^6$ and higher frequency than about 10^6. Presumably depends upon ϵ—the periodicity and patch size.

Form an equilibrium of $\vec{M}$ constant in space, then find the indices corresponding to patch interior points.

```
Me = 1+0.2*rand(1,3);
Me = Me./sqrt(sum(Me.^2,2))
Me = Me +0*patches.x;
Me([1 end],:,:,:)=nan;
i=find(~isnan(Me));
f0 = patchSys1(0,Me(:));
assert(abs( norm(f0(:)) )<1e-8,'not equilibrium')
```

Form the Jacobian by numerical differentiation.

```
delta=1e-7;
nJac=length(i);
Jac=nan(nJac);
for j=1:nJac
  M=Me; M(i(j))=M(i(j))+delta;
  fj=patchSys1(0,M(:));
  Jac(:,j)=(fj(i)-f0(i))/delta;
end
```

Compute eigenvalues, sort, and count some groups according to ad hoc criteria.

```
eval = eig(Jac);
[~,k] = sort(abs(eval));
eval = eval(k);
nZero = sum(abs(eval)<1)
nCent = sum(abs(real(eval))<1e5*alpha)
nSlow = sum(abs(eval)<1e5)
```

Plot the spectrum of eigenvalues on quasi-log axes.

```
figure(2),clf
hp = plot(real(eval),imag(eval),'.');
xlabel('$\Re\lambda$'), ylabel('$\Im\lambda$')
quasiLogAxes(hp,1,100);
ifOurCf2eps([mfilename 'Spec'])
```

3.10.3 `heteroLanLif1D()`: heterogeneous Landau–Lifshitz PDE

This function codes the lattice heterogeneous Landau–Lifshitz PDE (Leitenmaier & Runborg 2021, (1.1)) inside patches in 1D space. For 4D input array `M` storing the three components of $\vec{M}$ (via edge-value interpolation of `patchSys1`, Section 3.2), computes the time derivative at each point in the interior of a patch, output in `Mt`. The column vector of coefficients $c_i = 1 + \frac{1}{2}\sin(2\pi x_i/\epsilon)$ have previously been stored in struct `patches.cs`.

- With `ex5p1=0` computes the example EX1 (Leitenmaier & Runborg 2021, p.6).
- With `ex5p1=1` computes the first 'locally periodic' example (Leitenmaier & Runborg 2021, p.27).

```
29 function Mt = heteroLanLif1D(t,M,patches)
30   global alpha ex5p1
31   dx = diff(patches.x(2:3));   % space step
```

Compute the heterogeneous $\vec{H} := \vec{\nabla} \cdot (a\vec{\nabla}\vec{M})$

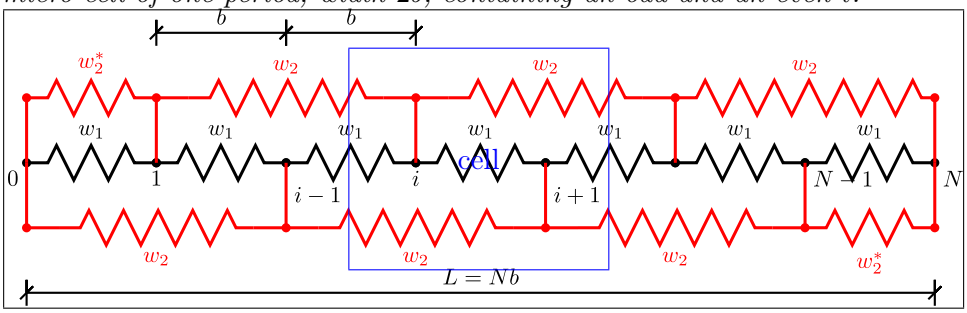
```
37   a = patches.cs ...
38      +ex5p1*(0.1+0.25*sin(2*pi*(patches.x(2:end,:,:,:)-dx/2)+1.1));
39   H = diff(a.*diff(M))/dx^2;
```

At each microscale grid point, compute the cross-products $\vec{M} \times \vec{H}$ and $\vec{M} \times (\vec{M} \times \vec{H})$ to then give the time derivative $\vec{M}_t = -\vec{M} \times \vec{H} - \alpha \vec{M} \times (\vec{M} \times \vec{H})$ (Leitenmaier & Runborg 2021, (1.1)):

```
47   i = 2:size(M,1)-1;   % interior points in a patch
48   MH=nan+H; % preallocate for MxH
49   MH(:,3,:,:) = M(i,1,:,:).*H(:,2,:,:)-M(i,2,:,:).*H(:,1,:,:);
50   MH(:,2,:,:) = M(i,3,:,:).*H(:,1,:,:)-M(i,1,:,:).*H(:,3,:,:);
51   MH(:,1,:,:) = M(i,2,:,:).*H(:,3,:,:)-M(i,3,:,:).*H(:,2,:,:);
52   MMH=nan+H; % preallocate for MxMxH
53   MMH(:,3,:,:)= M(i,1,:,:).*MH(:,2,:,:)-M(i,2,:,:).*MH(:,1,:,:);
54   MMH(:,2,:,:)= M(i,3,:,:).*MH(:,1,:,:)-M(i,1,:,:).*MH(:,3,:,:);
55   MMH(:,1,:,:)= M(i,2,:,:).*MH(:,3,:,:)-M(i,3,:,:).*MH(:,2,:,:);
56   Mt = nan+M; % preallocate output array
57   Mt(i,:,:,:) = -MH-alpha*MMH;
58 end% function
```

Fin.

Figure 3.15: 1D arrangement of non-linear springs with connections to (a) next-to-neighbour node (Combescure 2022, Fig. 3(a)). The blue box is one micro-cell of one period, width 2b, containing an odd and an even i.



3.11 `Combescure2022`: simulation and continuation of a 1D example nonlinear elasticity, via patches

Here we explore a nonlinear 1D elasticity problem with complicated microstructure. Executes a simulation. *But the main aim is to show how one may use the MatCont continuation toolbox (Govaerts et al. 2019) together with the Patch Scheme toolbox* (Maclean et al. 2020) in order to explore parameter space by continuing branches of equilibria, etc.

Figure 3.15 shows the microscale elasticity—adapted from Fig. 3(a) by Combescure (2022). Let the spatial microscale lattice be at rest at points x_i, with constant spacing b. With displacement variables $u_i(t)$, simulate the microscale lattice toy elasticity system with 2-periodicity: for $p = 1, 2$ (respectively black and red in Figure 3.15) and for every i,

$$\epsilon_i^p := \frac{1}{pb}(u_{i+p/2} - u_{i-p/2}), \qquad \sigma_i^p := w_p'(\epsilon_i^p),$$
$$\frac{\partial^2 u_i}{\partial t^2} = \sum_{p=1}^{2} \frac{1}{pb}(\sigma_{i+p/2}^p - \sigma_{i-p/2}^p), \qquad w_p'(\epsilon) := \epsilon - M_p\epsilon^3 + \epsilon^5. \tag{3.5}$$

The system has a microscale heterogeneity via the two different functions $w_p'(\epsilon)$ (Combescure 2022, §4):

- microscale 'instability' (structure) arises with $M_1 := 2$ and $M_2 := 1$ (Figures 3.16 and 3.19(b)); and
- large scale 'instability' (structure) arises with $M_1 := -1$ and $M_2 := 3$ (Figures 3.18 and 3.19(a)).

Microscale case Set $M_1 := 2$ and $M_2 := 1$. We fix the boundary conditions $u(0) = 0$ and parametrise solutions by $u(L)$. There are equilibria $u \approx u(L)x/L$, but under large compression (large negative $u(L)$) interesting structures develop. Figure 3.16 shows boundary layers with microscale variations develop for $u(L) < -13$. This figure plots a strain ϵ as the strain is nearly constant across the interior, so the boundary layers show up clearly. As $u(L)$ decreases further, Figure 3.16 shows the family of equilibria form complicated folds. Table 3.2 lists that MatCont also reports some branch

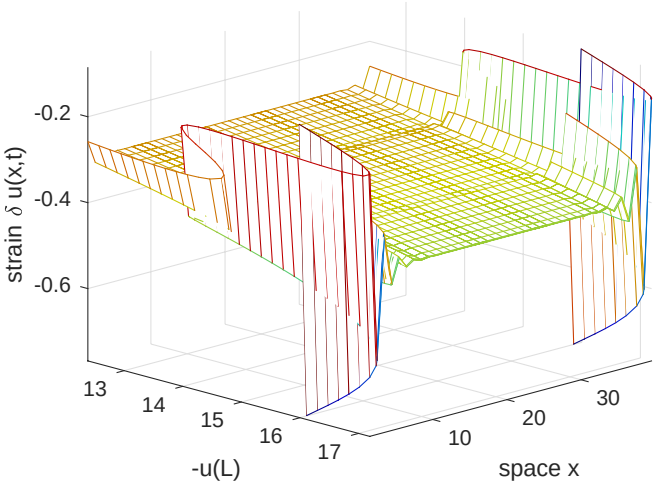


Figure 3.16: the case of microscale 'instability' appears as fluctuations close to both boundaries. As the system is physically compressed, the equilibrium curve has complicated folds, as shown here (and Figure 3.19(b)).

Table 3.2: Interesting equilibria for the cases of small scale instability: $M_1 := 2$, $M_2 := 1$ (Figures 3.16 and 3.19(b)). The rightmost column gives the $-u(L)$ parameter values for corresponding critical points in the three-patch code (Figure 3.17).

$-u(L)$	MatCont description	Patch
14.684	Branch point	14.599
14.702	Limit point	14.610
14.612	Neutral Saddle Equilibrium	-
14.063	Neutral Saddle Equilibrium	-
13.972	Limit point	13.817
13.988	Branch point	13.828
17.184	Branch point	17.197
-	Limit point	17.227
17.183	Neutral Saddle Equilibrium	17.211

points and neutral saddle equilibria in this same regime (see Figure 3.19(b)). I have not yet followed any of the branches.

The previous paragraph's discussion is for a full domain simulation, albeit done through an imposed computational framework of physically abutting patches. Figure 3.17 shows the corresponding MatCont continuation for the patch scheme with $N = 3$ patches in the domain. Just three patches may well be reasonable as the structures in this problem are the two boundary layers, and a constant interior. Figure 3.17 shows the patch scheme reasonably resolves these. Table 3.2 also lists the special points, as reported by MatCont, in the equilibria of the patch scheme. The locations of these special points reasonably match those found by the full domain simulation.

Importantly, MatCont is about *ten times quicker to execute on the patches* than on the full domain code. This speed-up indicates that on larger scale problems the patch scheme could be very useful in continuation explorations.

Large scale case Set $M_1 := -1$ and $M_2 := 3$. We fix the boundary conditions $u(0) = 0$ and parametrise solutions by $u(L)$. There are equilibria $u \approx u(L)x/L$, but under large compression (large negative $u(L)$) interesting structures develop. Figure 3.18 shows an interior region of higher magnitude strain develops. Again, this figure plots a strain ϵ as the strain is nearly constant across the domain, so the interior structure shows up clearly. As $u(L)$

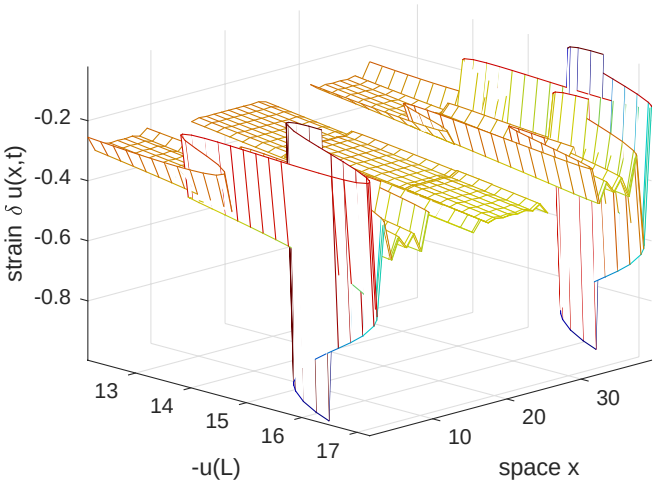


Figure 3.17: using just three patches, the case of microscale instability appears as fluctuations close to both boundaries. As the system is physically compressed, the equilibrium curve has complicated folds, as shown, and that approximately match Figure 3.16. But it is computed ten times quicker.

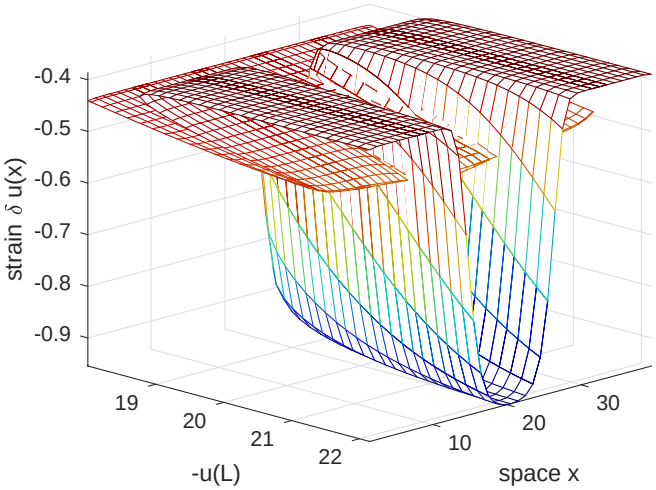


Figure 3.18: the case of large scale 'instability'. Spatial structure appears in the middle of the domain. As the system is physically compressed, the equilibrium curve has complicated folds, as shown here and in Figure 3.19(a).

decreases further, Figure 3.18 shows the family of equilibria form complicated folds. Table 3.3 lists that MatCont also reports some branch points and neutral saddle equilibria in this regime (see Figure 3.19(a)). I have not yet followed any of the branches.

The patch scheme with $N = 3$ patches does not make reasonable predictions here. I suspect this failure is because the nontrivial interior structure here occupies too much of the domain to fit into one 'small' patch. Here the patch scheme may be useful if the physical domain is larger.

3.11.1 Configure heterogeneous toy elasticity systems

Set some physical parameters. Each cell is of width $dx := 2b$ as I choose to store u_i for odd i in `u((i+1)/2,1,:)` and for even i in `u(i/2,2,:)`, that is,

Table 3.3: Interesting equilibria for the cases of large scale instability: $M_1 := -1$, $M_2 := 3$ (Figures 3.18 and 3.19(a)).

$-u(L)$	MatCont description
21.295	Limit point
18.783	Branch point
18.762	Neutral Saddle Equilibrium
18.761	Neutral Saddle Equilibrium
18.761	Limit point
18.934	Branch point
19.393	Branch point
19.928	Branch point
20.490	Branch point
21.055	Branch point
21.627	Branch point

the the physical displacements form the array

$$\mathtt{u} = \begin{bmatrix} u_1 & u_2 \\ u_3 & u_4 \\ u_5 & u_6 \\ \vdots & \vdots \end{bmatrix}.$$

Then corresponding velocities are adjoined as 3rd and 4th column.

```
clear all
global b M vis
b = 1   % separation of lattice points
N = 42  % # lattice steps in L
L = b*N % length of domain
```

The nonlinear coefficients of stress-strain are in array `M`, chosen by `theCase`.

```
theCase = 2
switch theCase
case 1, M = [0 0 0 0]  % linear spring coefficients
case 2, M = [ 2 1 1 1] % micro scale instability??
case 3, M = [-1 3 1 1] % large scale instability??
end% switch
vis = 0.1 % does not appear to affect the equilibria
tEnd = 25
```

Patch parameters: here `nSubP` is the number of cells.

```
edgyInt = true
nSubP = 6, nPatch = 5 % gives full-domain on N=42, dx=2
%nSubP = 6, nPatch = 3 % patches for some crude comparison
```

Establish the global data struct `patches` for the microscale heterogeneous lattice elasticity system (3.5). Solved with `nPatch` patches, and interpolation (as high-order as possible) to provide the edge-values of the inter-patch coupling conditions.

```
global patches
configPatches1(@heteroNLE,[0 L],'equispace',nPatch ...
```

```
270      ,0,2*b,nSubP,'EdgyInt',edgyInt);
271  xx = patches.x+[-1 1]*b/2; % staggered sub-cell positions
```

3.11.2 Simulate in time

Set the initial displacement and velocity of a simulation. Integrate some time using standard integrator.

```
284  u0 = [ sin(pi/L*xx)  -0*0.14*cos(pi/L*xx) ];
285  tic
286  [ts,ust] = ode23(@patchSys1, tEnd*linspace(0,1,41), u0(:) ...
287                   ,[],patches,0);
288  cpuIntegrateTime = toc
```

Plot space-time surface of the simulation To see the edge values of the patches, interpolate and then adjoin a row of `nan`s between patches. Because of the odd/even storage we need to do a lot of permuting and reshaping. First, array of sub-cell coordinates in a column for each patch, separating patches also by an extra row of nans.

```
301  xs = reshape( permute( xx ,[2 1 3 4]), 2*nSubP,nPatch);
302  xs(end+1,:) = nan;
```

Interpolate patch edge values, at all times simultaneously by including time data into the 2nd dimension, and 2nd reshaping it into the 3rd dimension.

```
310  uvs = reshape( permute( reshape(ust ...
311        ,length(ts),nSubP,4,1,nPatch) ,[2 3 1 4 5]) ,nSubP,[],1,nPatch);
312  uvs = reshape( patchEdgeInt1(uvs) ,nSubP,4,[],nPatch);
```

Extract displacement field, merge the 1st two columns, permute the time variations to the 3rd, separate patches by NaNs, and merge spatial data into the 1st column.

```
320  us = reshape( permute( uvs(:,1:2,:,:) ...
321       ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
322  us(end+1,:,:) = nan;
323  us = reshape(us,[],length(ts));
```

Plot space-time surface of displacements over the macroscale duration of the simulation.

```
330    figure(1), clf()
331    mesh(ts,xs(:),us)
332    view(60,40), colormap(0.8*jet), axis tight
333    xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
```

Ditto for the velocity.

```
339  vs = reshape( permute( uvs(:,3:4,:,:) ...
340       ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
341  vs(end+1,:,:) = nan;
342  vs = reshape(vs,[],length(ts));
343    figure(2), clf()
```

```
344   mesh(ts,xs(:),vs)
345   view(60,40), colormap(0.8*jet), axis tight
346   xlabel('time $t$'), ylabel('space $x$'), zlabel('$v(x,t)$')
347   drawnow
```

3.11.3 MatCont continuation

First, use `fsolve` to find an equilibrium at some starting compressive displacement—a compression that differs depending upon the case of nonlinearity.

```
359 muL0 = 12+6*(theCase==3)
360 u0 = [ -muL0*xx/L 0*xx ];
361 u0([1 end],:,:,:)=nan;
362 patches.i = find(~isnan(u0));
363 nVars=length(patches.i)
364 ueq=fsolve(@(v) dudtSys(0,v,muL0),u0(patches.i));
```

Start search for equilibria at other compression parameters. Starting from zero, need 1000+ to find both the large-scale and small-scale instability cases. But need less points when starting from parameter 12 or so.

```
373 disp('Searching for equilibria, may take 1000+ secs')
374 [uv0,vec0]=init_EP_EP(@matContSys,ueq,muL0,[1]);
375 opt=contset; % initialise MatCont options
376 opt=contset(opt,'Singularities',true); %to report branch points, p.24
377 opt=contset(opt,'MaxNumPoints',400); % restricts how far matcont goes
378 opt=contset(opt,'Backward',true); % strangely, needs to go backwards??
379 [uv,vec,s,h,f]=cont(@equilibrium, uv0, [], opt); %MatCont continuation
```

Post-process the report

```
386 disp('List of interesting critical points')
387 muLs=uv(nVars+1,:);
388 for j=1:numel(s)
389    disp([num2str(muLs(s(j).index),5) ' & ' s(j).msg ' \\'])
390 end
```

Find a range of parameter and corresponding indices where all the critical points occur.

```
397 p1=muLs(end); pe=muLs(1);
398 if numel(s)>3, for j=2:numel(s)-1
399     p1=min(p1,muLs(s(j).index));
400     pe=max(pe,muLs(s(j).index));
401 end, end
402 pMid=(p1+pe)/2, pWid=abs(pe-p1)
403 iPars=find(abs(muLs(:)-pMid)<pWid);%include some either side
```

Choose an 'evenly spaced' subset of the range so we only plot up to sixty of the parameter values reported in the range.

```
411 nPars=numel(iPars)
412 dP=ceil((nPars-1)/60)
```

```
iP=1:dP:nPars;
muLP=muLs(iPars(iP));
```

Interpolate patch edge values, at all parameters simultaneously by including parameter-wise data into the 2nd dimension, and 2nd reshaping it into the 3rd dimension.

```
uvs=nan(numel(iP),numel(u0));
uvs(:,patches.i)=uv(1:nVars,iPars(iP))';
uvs = reshape( permute( reshape(uvs ...
      ,length(muLP),nSubP,4,1,nPatch) ,[2 3 1 4 5]) ,nSubP,[],1,nPatch);
uvs = reshape( patchEdgeInt1(uvs) ,nSubP,4,[],nPatch);
```

Extract displacement field, merge the 1st two columns, permute the parameter variations to the 3rd, separate patches by NaNs, and merge spatial data into the 1st column.

```
us = reshape( permute( uvs(:,1:2,:,:) ...
     ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
us(end+1,:,:) = nan;
us = reshape(us,[],length(muLP));
```

Plot space-time surface of displacements over the macroscale duration of the simulation.

```
  figure(4), clf()
  mesh(muLP,xs(:),us)
  view(60,40), colormap(0.8*jet), axis tight
  xlabel('-u(L)'), ylabel('space $x$'), zlabel('$u(x)$')
```

Plot space-time surface of strain, differences in displacements, over the parameter variation.

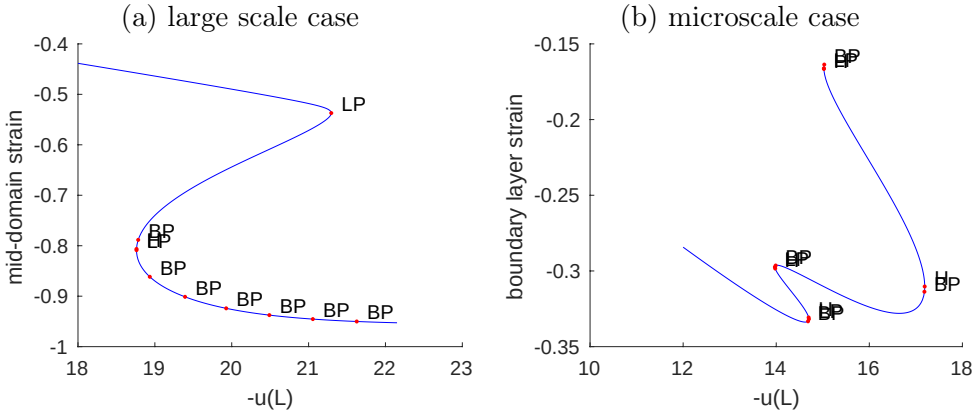
```
  figure(5), clf()
  mesh(muLP,xs(1:end-1),diff(us))
  view(45,20), colormap(0.8*jet), axis tight
  xlabel('$-u(L)$'), ylabel('space $x$'), zlabel('strain $\delta u(x)$')
  ifOurCf2eps(['Comb22diffu' num2str(theCase)],[12 9])%optionally save
```

Labelled parameter plot Get the labelled 2D plots of Figure 3.19 via MatCont's `cpl` function. In high-D problems it is unlikely that any one variable is a good thing to plot, so I show how to plot something else, here a strain. I use all the computed points so reform `uvs` (possibly better to have merged the critical points into the list of plotted parameters??).

```
uvs = nan(numel(muLs),numel(u0));
uvs(:,patches.i) = uv(1:nVars,:)';
uvs = reshape( uvs ,[],nSubP,4,nPatch);
```

As a function of the parameter, plot the strain in the middle of the domain (the middle of the middle patch), unless it is the microscale case when we plot a strain near the middle of the left boundary layer.

Figure 3.19: cross-sections through Figures 3.16 and 3.18: (a) large scale case, at the mid-point in space of Figure 3.18; (b) microscale case, in a boundary layer of Figure 3.16. These cross-sections are labelled with the various critical points.




```
499 if theCase==2, thePatch=1;
500 else thePatch=(nPatch+1)/2;
501 end%if
502   figure(7),clf
503   du = diff( uvs(:,nSubP/2,1:2,thePatch) ,1,3);
504   cpl([muLs;du'],[],s);
505   xlabel('$-u(L)$')
506   if thePatch==1,  ylabel('boundary layer strain')
507   else ylabel('mid-domain strain')
508   end
509   ifOurCf2eps(['Comb22cpl' num2str(theCase)],[9 7])%optionally save
```

3.11.4 `matContSys`: basic function for MatCont analysis

This is the simple 'odefile' of the patch scheme wrapped around the microcode.

```
522 function out = matContSys%(t,coordinates,flag,y,z)
523 out{1} = [];%@init;
524 out{2} = @dudtSys;
525 out{3} = [];%@jacobian;
526 out{4} = [];%@jacobianp;
527 out{5} = [];%@hessians;
528 out{6} = [];%@hessiansp;
529 out{7} = [];
530 out{8} = [];
531 out{9} = [];
532 end% function matContSys
```

3.11.5 `dudtSys()`: wraps around the patch wrapper

This function adjoins `patches` to the argument list, places the variables within the patch structure, and then extracts their time derivatives to return. Used

by both MatCont and `fsolve`.

```
545 function ut = dudtSys(t,u,p)
546 global patches
```

The 4 here is the number of variables in each micro-cell, that is, notionally 'at' each x-grid point.

```
553 U=nan(1,4)+patches.x;
554 U(patches.i)=u(:);
555 Ut=patchSys1(t,U,patches,p);
556 ut=Ut(patches.i);
557 end
```

3.11.6 `heteroNLE()`: forced heterogeneous elasticity

This function codes the lattice heterogeneous example elasticity inside the patches. Computes the time derivative at each point in the interior of a patch, output in `uvt`.

```
573 function uvt = heteroNLE(t,uv,patches,muL)
574   if nargin<4, muL=0; end% default end displacement is zero
575   global b M vis
```

Separate state vector into displacement and velocity fields: u_{ijI} is the displacement at the jth point in the ith 2-cell in the Ith patch; similarly for velocity v_{ijI}. That is, physically neighbouring points have different j, whereas physical next-to-neighbours have i different by one.

```
586   u=uv(:,1:2,:,:); v=uv(:,3:4,:,:); % separate u and v=du/dt
```

Provide boundary conditions, here fixed displacement and velocity in the left/right sub-cells of the leftmost/rightmost patches.

```
595 u(1,:,:,1)=0;
596 v(1,:,:,1)=0;
597 u(end,:,:,end)=-muL;
598 v(end,:,:,end)=0;
```

Compute the two different strain fields, and also a first derivative for some optional viscosity.

```
606   eps2 = diff(u)/(2*b);
607   eps1 = [u(:,2,:,:)-u(:,1,:,:) u([2:end 1],1,:,:)-u(:,2,:,:)]/b;
608   eps1(end,2,:,:)=nan; % as this value is fake
609   vx1  = [v(:,2,:,:)-v(:,1,:,:) v([2:end 1],1,:,:)-v(:,2,:,:)]/b;
610   vx1(end,2,:,:)=nan; % as this value is fake
```

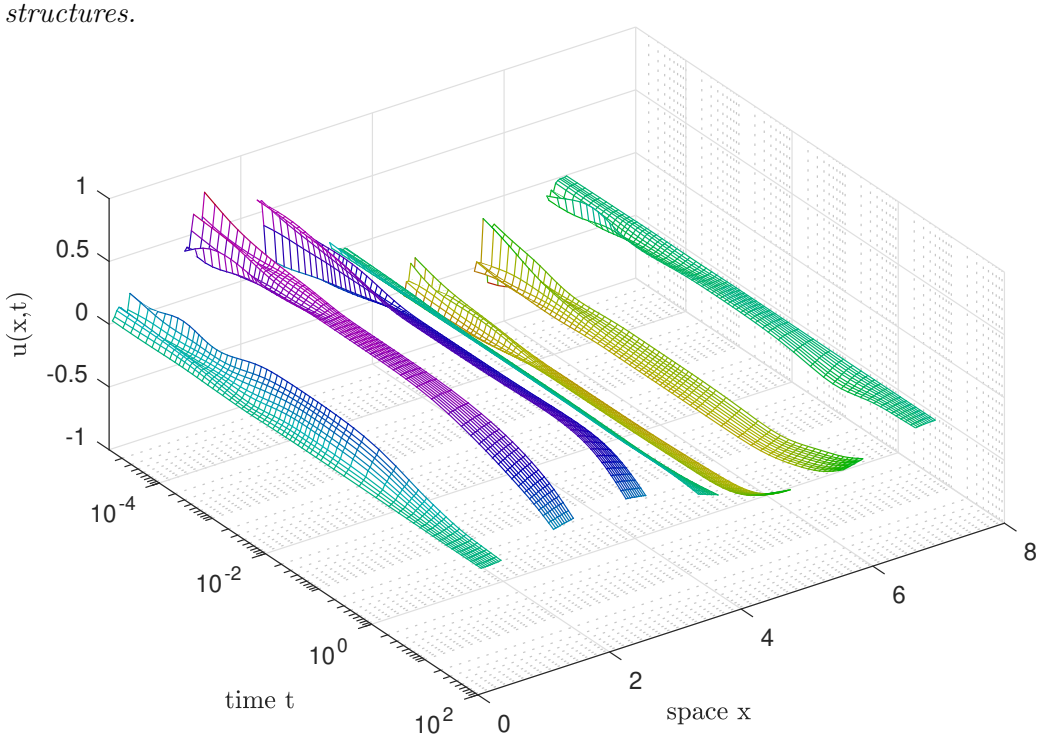
Set corresponding nonlinear stresses

```
616   sig2 = eps2-M(2)*eps2.^3+M(4)*eps2.^5;
617   sig1 = eps1-M(1)*eps1.^3+M(3)*eps1.^5;
```

Preallocate output array, and fill in time derivatives of displacement and velocity, from velocity and gradient of stresses, respectively.

```
625   uvt = nan+uv;          % preallocate output array
626   i=2:size(uv,1)-1;
627   % rate of change of position
628   uvt(i,1:2,:,:) = v(i,:,:,:);
629   % rate of change of velocity +some artificial viscosity??
630   uvt(i,3:4,:,:) = diff(sig2) ...
631     +[ sig1(i,1,:,:)-sig1(i-1,2,:,:)  diff(sig1(i,:,:,:),1,2)] ...
632   +vis*[ vx1(i,1,:,:)-vx1(i-1,2,:,:)  diff(vx1(i,:,:,:),1,2) ];
633
634 end% function heteroNLE
```

Figure 3.20: hyper-diffusing field $u(x,t)$ in the patch scheme applied to microscale heterogeneous hyper-diffusion (Section 3.12). The log-time axis shows: $t < 10^{-2}$, rapid decay of sub-patch micro-structure; $10^{-2} < t < 1$, meso-time quasi-equilibrium; and $1 < t < 10^2$, slow decay of macroscale structures.



3.12 `hyperDiffHetero`: simulate a heterogeneous hyper-diffusion PDE in 1D on patches

Section contents

Figure 3.20 shows an example simulation in time generated by the patch scheme applied to a heterogeneous version of the hyper-diffusion PDE. That such simulations makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

We aim to simulate the heterogeneous hyper-diffusion PDE

$$u_t = -D[c_1(x)Du] \quad \text{where operator } D := \partial_x(c_2(x)\partial_x), \tag{3.6}$$

for microscale periodic coefficients $c_l(x)$, and boundary conditions of $u = u_x = 0$ at $x = 0, L$. In this 1D space, the macroscale, homogenised, effective hyper-diffusion should be some unknown 'average' of these coefficients, but we use the patch scheme to provide a computational homogenisation. We discretise the PDE to a lattice of values $u_i(t)$, with lattice spacing dx, and

governed by

$$\dot{u}_i = -D[c_{i1} D u_i] \quad \text{where operator } D := \delta(c_{i2}\delta)/dx^2$$

in terms of centred difference operator $\delta u_i := u_{i+1/2} - u_{i-1/2}$.

Set the desired microscale periodicity, and correspondingly choose random microscale diffusion coefficients (with some subscripts shifted by a half).

```
clear all
basename = mfilename
%global OurCf2eps, OurCf2eps=true %optional to save plots
nGap = 3 % controls size of gap between patches
nPtsPeriod = 5
dx = 0.5/nGap/nPtsPeriod
```

Create some random heterogeneous coefficients, log-uniform.

```
csVar = 1
cs = 0.2*exp( -csVar/2+csVar.*rand(nPtsPeriod,2) )
```

Establish global data struct `patches` for heterogeneous hyper-diffusion on a finite domain with, on average, one patch per unit length. Use seven patches, and use high-order interpolation with `ordCC` $= 0$.

```
nPatch = 7
nSubP = 2*nPtsPeriod+4 % or +2 for not-edgyInt
Len = nPatch;
ordCC = 0;
dom.type = 'equispace';
dom.bcOffset = 0.5 % for BC type
patches = configPatches1(@hyperDiffPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
    ,'hetCoeffs',cs);
xs=squeeze(patches.x);
```

Simulate in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 3.2) to the microscale differential equations.

```
u0 = sin(2*pi/Len*patches.x).*rand(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 100], u0(:) ,[],patches);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 3.20, using log-axis for time so we can see a little of both micro- and macro-dynamics.

```
figure(1),clf
xs([1:2 end-1:end],:) = nan;
t0=min(find(ts>1e-5));
mesh(ts(t0:3:end),xs(:),us(t0:3:end,:)'),  view(55,50)
```

```
colormap(0.7*hsv)
xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
ca=gca;  ca.XScale='log';  ca.XLim=ts([t0 end]);
ifOurCf2eps([basename 'Uxt'])
```

Fin.

3.12.1 Heterogeneous hyper-diffusion PDE inside patches

As a microscale discretisation of hyper-diffusion PDE (3.6) $u_t = -D[c_1(x)Du]$, where heterogeneous operator $D = \partial_x(c_2(x)\partial_x)$.

```
function ut=hyperDiffPDE(t,u,patches)
  dx=diff(patches.x(1:2));  % microscale spacing
```

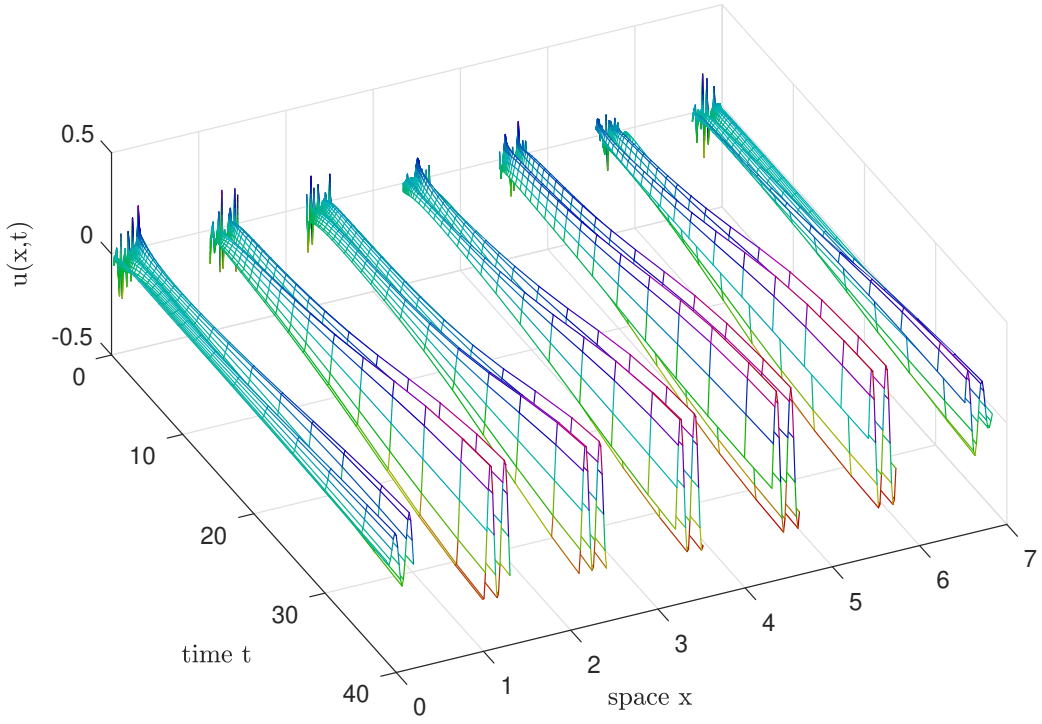
Code Dirichlet boundary conditions of zero function and derivative at left-end of left-patch, and right-end of right-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  if ~patches.periodic % discretise BC u=u_x=0
      u(1:2,1)=0;
      u(end-1:end,end)=0;
  end%if
```

Here code straightforward centred discretisation in space.

```
  ut = nan+u;   % preallocate output array
  v = patches.cs(2:end,1).*diff(patches.cs(:,2).*diff(u))/dx^2;
  ut(3:end-2,:) = -diff(patches.cs(2:end-1,2).*diff(v))/dx^2 ;
end
```

Figure 3.21: the pattern forming field $u(x,t)$ in the patch (gap-tooth) scheme applied to a microscale discretisation of the Swift–Hohenberg PDE *(Section 3.13). Physically we see the rapid decay of much microstructure, but also the meso-time growth of sub-patch-scale patterns, wavenumber k_0, that are modulated over the inter-patch distances and over long times.*



3.13 `SwiftHohenbergPattern`: patterns of the Swift–Hohenberg PDE in 1D on patches

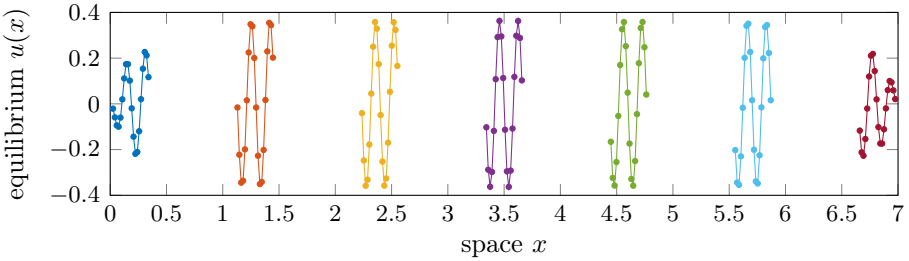
Section contents

Figure 3.21 shows an example simulation in time generated by the patch scheme applied to the patterns arising from the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is just more than a multiple of the periodicity of the pattern.

Consider a lattice of values $u_i(t)$, with lattice spacing dx, and governed by a microscale centred discretisation of the Swift–Hohenberg PDE

$$\partial_t u = -(1 + \partial_x^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3, \tag{3.7}$$

Figure 3.22: an equilibrium of the Swift–Hohenberg PDE *on seven patches in 1D space. In the sub-patch patterns, there is a small phase shift in the patterns from patch to patch. And the amplitude of the pattern has to go to 'zero' at the boundaries.*



with boundary conditions of $u = u_x = 0$ at $x = 0, L$. For Ra just above critical, say Ra $= 0.1$, the system rapidly evolves to spatial quasi-periodic solutions with period ≈ 0.166 when wavenumber parameter $k_0 = 38$. On medium times these spatial oscillations grow to near equilibrium amplitude of $\sqrt{\text{Ra}}$, and over very long times the phases of the oscillations evolve in space to adapt to the boundaries.

Set the desired microscale periodicity of the emergent pattern.

```
clear all, close all
%global OurCf2eps, OurCf2eps=true %optional to save plots
Ra = 0.1 % Ra>0 leads to patterns
nGap = 3
%waveLength = 0.496688741721854 /nGap %for nPatch==5
waveLength = 0.497630331753555 /nGap %for nPatch==7
%waveLength = 0.5 /nGap %for periodic case
nPtsPeriod = 10
dx = waveLength/nPtsPeriod
k0 = 2*pi/waveLength
```

Establish global data struct `patches` for the Swift–Hohenberg PDE on some length domain. Use seven patches. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions.

```
nPatch = 7
nSubP = 2*nPtsPeriod+4
%nSubP = 2*nGap*nPtsPeriod+4 % full-domain
Len = nPatch;
ordCC = 4;
dom.type='equispace';
dom.bcOffset=0.5
patches = configPatches1(@SwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
xs=squeeze(patches.x);
```

3.13.0.1 Find equilibrium with fsolve

Start the search from some guess.

```
fprintf('\n**** Find equilibrium with fsolve\n')
u = 0.4*sin(k0*patches.x);
```

But set the pairs of patch-edge values to `Nan` in order to use `patches.i` to index the interior sub-patch points as they are the variables.

```
u([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u));
```

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 3.15).

```
tic
[u(patches.i),res] = fsolve(@(v) theRes(v,patches,k0,Ra) ...
    ,u(patches.i) ,optimoptions('fsolve','Display','off'));
solveTime = toc
normRes = norm(res)
assert(normRes<1e-6,'**** fsolve solution not accurate')
```

Plot the equilibrium see Figure 3.22.

```
figure(1),clf
subplot(2,1,1)
plot(xs,squeeze(u),'.-')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([mfilename 'Equilib'])%optionally save
```

3.13.0.2 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 3.2) to the microscale differential equations.

```
fprintf('\n**** Simulate in time\n')
u0 = 0*patches.x+0.1*randn(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 40], u0(:) ,[],patches,k0,Ra);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 3.21.

```
figure(2),clf
xs([1:2 end-1:end],:) = nan;
mesh(ts(1:3:end),xs(:),us(1:3:end,:)'),  view(65,60)
colormap(0.7*hsv)
xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
ifOurCf2eps([mfilename 'Uxt'])
```

Fin.

3.13.1 The Swift–Hohenberg PDE and BCs inside patches

As a microscale discretisation of Swift–Hohenberg PDE $u_t = -(1+\partial_x^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3$, here code straightforward centred discretisation in space.

```
function ut=SwiftHohenbergPDE(t,u,patches,k0,Ra)
  dx=diff(patches.x(1:2));  % microscale spacing
  i=3:size(u,1)-2;  % interior points in patches
```

Code Dirichlet boundary conditions of zero function and derivative, $u = u_x = 0$, at the left-end of the leftmost-patch, and the right-end of the rightmost-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  u(1:2,1)=0;
  u(end-1:end,end)=0;
```

Here code straightforward centred discretisation in space.

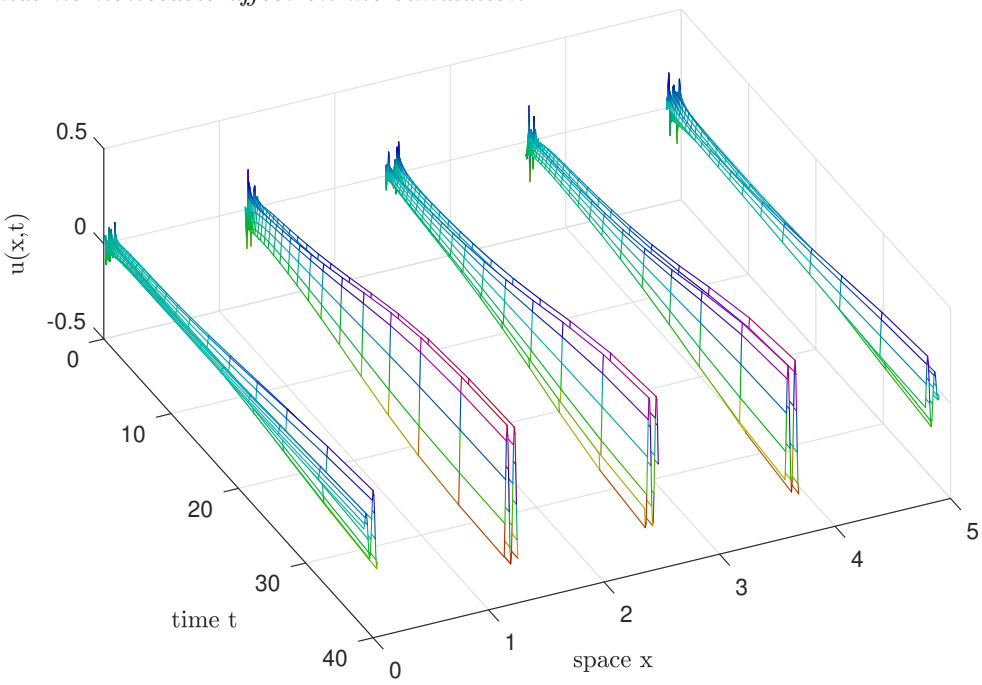
```
  ut=nan+u;         % preallocate output array
  v = u(2:end-1,:)+diff(u,2)/dx^2/k0^2;
  ut(i,:) = -( v(2:end-1,:)+diff(v,2)/dx^2/k0^2 ) ...
    +Ra*u(i,:) -u(i,:).^3;
end
```

3.13.2 `theRes()`: wrapper function to zero for equilibria

This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time zero, and returns the vector of patch-interior time derivatives.

```
function f=theRes(u,patches,k0,Ra)
  v=nan(size(patches.x));
  v(patches.i) = u;
  f = patchSys1(0,v(:),patches,k0,Ra);
  f = f(patches.i);
end%function theRes
```

Figure 3.23: the field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous Swift–Hohenberg PDE *(Section 3.14). The heterogeneous coefficients are approximately uniform over $[0.9, 1.1]$. This heterogeneity has no noticeable affect on the simulation.*



3.14 `SwiftHohenbergHetero`: patterns of a heterogeneous Swift–Hohenberg PDE in 1D on patches

Section contents

Figure 3.23 shows an example simulation in time generated by the patch scheme applied to the patterns arising from a heterogeneous version of the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

Consider a lattice of values $u_i(t)$, with lattice spacing dx, arising from a microscale discretisation of the pattern forming, heterogeneous, Swift–Hohenberg PDE

$$\partial_t u = -D[c_1(x)Du] + \mathrm{Ra}\, u - u^3, \quad D := 1 + \partial_x[c_2(x)\partial_x \cdot]/k_0^2, \tag{3.8}$$

where $c_\ell(x)$ have period $2\pi/k_0$. Coefficients c_ℓ are chosen iid random, nearly uniform, with mean near one. With mean one, the periodicity of c_ℓ approximately matches the periodicity of the resultant spatial pattern.

The current patch scheme coding preserves symmetry in the case of periodic patches (for every order of interpolation). For equispace and chebyshev options, the coupling currently fails symmetry.

Consider the spectrum in the symmetric cases of periodic patches (based upon only the cases $N = 5, 7$). There are $2N$ small eigenvalues, separated by a gap from the rest. In the homogeneous case, these occur as N pairs. With small heterogeneity, they appear to split into $N - 1$ pairs, and two distinct. With stronger heterogeneity (say 0.5), they *often* appear to also split into two clusters, each of N eigenvalues, with one small-valued cluster, and one meso-valued cluster—curious. Further analysis with sparse approximation of the invariant spaces suggests the following:

- for homogeneous, the $2N$ modes are local oscillations in each patch, with two modes each corresponding to phase shifts of the possible oscillations;
- for heterogeneous
  - N eigenmodes appear to be one phase 'locking' to the heterogeneity; and
  - N eigenmodes appear to be other phase 'locking' to the heterogeneity. Unless it is something to do with the coupling, but then it only appears with heterogeneity.

Consider the spectrum with BCs of $u = u_{xx} = 0$ at ends. Non-symmetric so some eigenvalues are complex! For small or zero heterogeneity find $2N - 2$ eigenvalues are small. Effectively, two modes in each of $N - 2$ interior patches, and one mode each in the two end patches. With increasing heterogeneity (say above 0.3), the gap decreases as a couple (or some) of the small eigenvalues become larger in magnitude.

Consider the spectrum with BCs of $u = u_x = 0$ at ends. Non-symmetric so some eigenvalues are complex! For small or zero heterogeneity find $2N - 4$ eigenvalues are small. Effectively, two modes in each of $N - 2$ interior patches. With increasing heterogeneity (say above 0.4), half $(N - 2)$ of the small eigenvalues become larger in magnitude (presumably some phase 'locking' to the heterogeneity): effectively forms two clusters of modes.

Set the desired microscale periodicity of the patterns, here 0.062, and on the microscale lattice of spacing 0.0062, correspondingly choose random microscale material coefficients. The wavenumber of this microscale patterns is $k_0 \approx 101$.

```
clear all
%global OurCf2eps, OurCf2eps=true %optional to save plots
basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
Ra = 0.1  % Ra>0 leads to patterns
nGap = 8 % controls size of gap between patches
```

```
waveLength = 0.496688741721854 /nGap %for nPatch==5
%waveLength = 0.497630331753555 /nGap %for nPatch==7
%waveLength = 0.5 /nGap %for periodic case
nPtsPeriod = 10
dx = waveLength/nPtsPeriod
k0 = 2*pi/waveLength
```

Create some random heterogeneous coefficients.

```
heteroVar = 0.99*[1 1] % must be <2
cl = 1./(1-heteroVar/2+heteroVar.*rand(nPtsPeriod,2));
cRange = quantile(cl,0:0.5:1)
```

Establish global data struct `patches` for heterogeneous Swift–Hohenberg PDE with, on average, one patch per units length. Use seven patches to start with. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Or use as high-order as possible with `ordCC` $= 0$.

```
nPatch = 5
nSubP = 2*nPtsPeriod+4 % +2 for not-edgyInt
%nSubP = 2*nGap*nPtsPeriod+4 % approx full-domain
Len = nPatch;
ordCC = 0;
dom.type='equispace';
dom.bcOffset=0.5
patches = configPatches1(@heteroSwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
    ,'hetCoeffs',cl);
xs=squeeze(patches.x);
```

3.14.0.1 Explore the Jacobian

Finds that with periodic patches, everything is symmetric. However, for equispace or chebyshev, the patch coupling is not symmetric—is this to be expected?

```
fprintf('\n**** Explore the Jacobian\n')
u0 = 0*patches.x;
u0([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u0));
nVars = numel(patches.i)
Jac = nan(nVars);
for j=1:nVars
    Jac(:,j)=theRes((1:nVars)==j,patches,k0,0,0);
end
```

Check on the symmetry of the Jacobian

```
nonSymmetric = norm(Jac-Jac')
Jac(abs(Jac)<1e-12)=0;
antiJac = Jac-Jac';
```

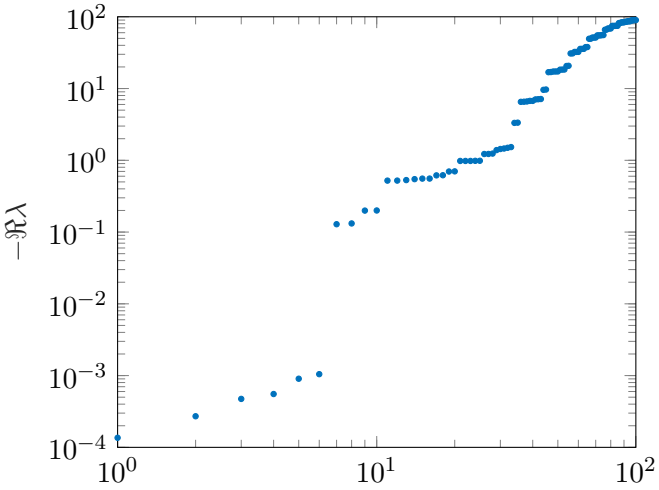


Figure 3.24: eigenvalues of the patch scheme on the heterogeneous Swift–Hohenberg PDE (linearised). With $N = 5$ patches and BCs of $u = u_x = 0$ at $x \in \{0, 5\}$, there are $2(N - 2) = 6$ small eigenvalues, $|\lambda| < 0.001$, corresponding to six slow modes in the interior.

```
antiJac(abs(antiJac)<1e-12)=0;
figure(6),clf
spy(Jac,'.'),hold on, spy(antiJac,'rx'),hold off
if nonSymmetric>5e-9, warning('failed symmetry'),
else Jac = (Jac+Jac')/2; %tweak to symmetry
end
```

Compute eigenvalues and eigenvectors.

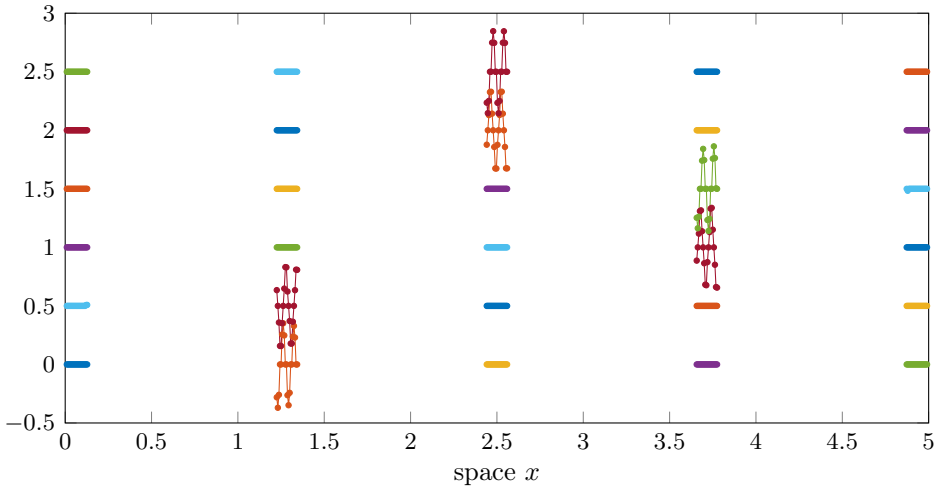
```
figure(5),clf
[evec,mEval] = eig(-Jac ,'vector');
[~,j]=sort(real(mEval));
mEval=mEval(j); evec=evec(:,j);
loglog(real(mEval),'.')
ylabel('$-\Re\lambda$')
ifOurCf2tex([basename 'Eval'])%optionally save
```

Explore sparse approximations of all the slowest together (lots of iterations required), or separately of the two clusters of the slowest (few iterations needed). First ascertain whether one or two clusters of small eigenvalues.

```
logGaps=diff(log10(real(mEval)));
[~,j]=sort(-logGaps);
%someLogGaps=[logGaps(j(1:5)) j(1:5)]
if logGaps(j(2))<0.4*logGaps(j(1)), nSlow=j(1)
else nSlow=min( sort(j(1:2)) , 3*nPatch)
end
log10Gap=logGaps(nSlow)
smallEvals=-mEval(1:nSlow(end)+2)
```

Second, make eigenvectors all real, sparsely approximate cluster modes via an algorithm developed from Hu et al. (2016), and plot. Figure 3.25 shows that each pair of basis vectors are phase-shifted by 90°.

Figure 3.25: sparse approximations of the eigenvectors of the six slow modes of Figure 3.24. Plotted are sparse basis vectors for the invariant space spanned by the six slow eigenvectors: each basis vector shifted vertically to separate. Thus a fair approximation is that there are effectively two modes for each of the $N - 2 = 3$ interior patches.

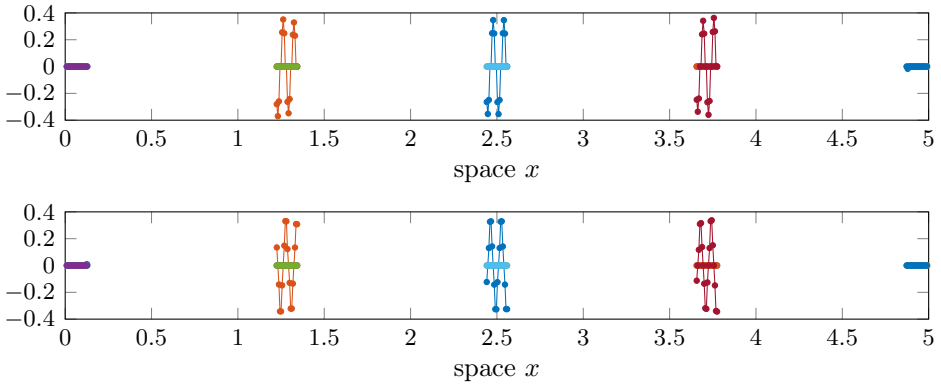



```
js=find(imag(mEval)>0);
evec(:,js)=imag(evec(:,js));
evec=real(evec);
if numel(nSlow)==1, S = spcart(evec(:,1:nSlow));
else  S = spcart(evec(:,1:nSlow(1)));
   S = [S spcart(evec(:,nSlow(1)+1:nSlow(2))) ];
end;
figure(3),clf
vStep=ceil(max(abs(S(:)))*10+1)/10
for j=1:nSlow(end)
  u0(patches.i)=S(:,j);
  plot(xs,vStep*(j-1)+squeeze(u0),'.-'),hold on
end
hold off, xlabel('space $x$')
ifOurCf2tex([basename 'Evec'])%optionally save
```

Reorganise the eigenvectors to maybe clarify.

```
[i,j]=find(abs(S)>vStep/2);
j=find([1;diff(j)]);
[i,k]=sort(i(j));
figure(4)
for p=1:2
  clf,subplot(2,1,1)
  for j=p:2:numel(k)
    u0(patches.i)=S(:,k(j));
    plot(xs,squeeze(u0),'.-'),hold on
  end% for j
  hold off, xlabel('space $x$')
```

Figure 3.26: sparse basis approximations for the invariant subspace of the six slow modes of Figure 3.24. A replot of Figure 3.25 but with three of the basis vectors superimposed in each of the two panels.




```
  ifOurCf2tex([basename 'Evec' num2str(p)])%optionally save
end%for p
```

3.14.0.2 Find an equilibrium with fsolve

Start the search from some guess.

```
fprintf('\n**** Find equilibrium with fsolve\n')
u = 0.4*sin(2*pi/waveLength*patches.x);
```

But set the pairs of patch-edge values to `Nan` in order to use `patches.i` to index the interior sub-patch points as they are the variables.

```
u([1:2 end-1:end],:) = nan;
patches.i = find(~isnan(u));
```

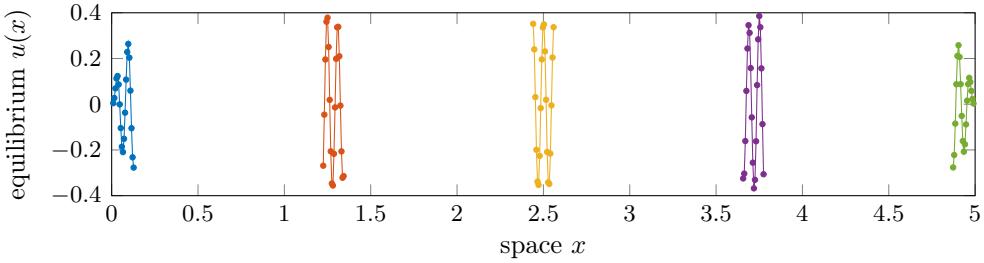
Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 3.14.2).

```
tic
[u(patches.i),res] = fsolve(@(v) theRes(v,patches,k0,Ra,1) ...
    ,u(patches.i) ,optimoptions('fsolve','Display','off'));
solveTime = toc
normRes = norm(res)
if normRes>1e-7, warning('residual large: bad equilibrium'),end
```

Plot the equilibrium see Figure 3.27.

```
figure(1),clf
subplot(2,1,1)
plot(xs,squeeze(u),'.-')
xlabel('space $x$'),ylabel('equilibrium $u(x)$')
ifOurCf2tex([basename 'Equilib'])%optionally save
```

Figure 3.27: an equilibrium of the heterogeneous Swift–Hohenberg PDE *determined by the patch scheme*



3.14.0.3 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSys1` (Section 3.2) to the microscale differential equations.

```
fprintf('\n**** Simulate in time\n')
u0 = 0*sin(2*pi/waveLength*patches.x)+0.1*randn(nSubP,1,1,nPatch);
tic
[ts,us] = ode15s(@patchSys1, [0 40], u0(:) ,[],patches,k0,Ra,1);
simulateTime = toc
us = reshape(us,length(ts),numel(patches.x(:)),[]);
```

Plot the simulation in Figure 3.23.

```
figure(2),clf
xs([1:2 end-1:end],:) = nan;
mesh(ts(1:3:end),xs(:),us(1:3:end,:)'),  view(65,60)
colormap(0.7*hsv)
xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
ifOurCf2eps([basename 'Uxt'])
```

Fin.

3.14.1 Heterogeneous SwiftHohenberg PDE+BCs inside patches

As a microscale discretisation of Swift–Hohenberg PDE $u_t = -D[c_1(x)Du] + \mathrm{Ra}\,u - u^3$, where heterogeneous operator $D = 1 + \partial_x(c_2(x)\partial_x)/k_0^2$.

```
function ut=heteroSwiftHohenbergPDE(t,u,patches,k0,Ra,cubic)
  dx=diff(patches.x(1:2));  % microscale spacing
  i=3:size(u,1)-2;  % interior points in patches
```

Code a couple of different boundary conditions of zero function and derivative(s) at left-end of left-patch, and right-end of right-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

```
  u = squeeze(u);
  if ~patches.periodic
    switch 1
```

```
402       case 1 % these are u=u_x=0
403         u(1:2,1)=0;
404         u(end-1:end,end)=0;
405       case 2 % these are u=u_{xx}=0
406         u(1:2,1) = [-u(3,1); 0];
407         u(end-1:end,end) = [0; -u(end-2,end)];
408       end% case
409     end%if
```

Here code straightforward centred discretisation in space.

```
415     ut = nan+u;          % preallocate output array
416     v = u(2:end-1,:)+diff(patches.cs(:      ,2).*diff(u))/dx^2/k0^2;
417     v = v.*patches.cs(2:end,1);
418     v = v(2:end-1,:)+diff(patches.cs(2:end-1,2).*diff(v))/dx^2/k0^2;
419     ut(i,:) = -v +Ra*u(i,:) -cubic*u(i,:).^3;
420   end
```

3.14.2 `theRes()`: a wrapper function

This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time zero, and returns the vector of patch-interior time derivatives.

```
435   function f=theRes(u,patches,k0,Ra,cubic)
436     v=nan(size(patches.x));
437     v(patches.i) = u;
438     f = patchSys1(0,v(:),patches,k0,Ra,cubic);
439     f = f(patches.i);
440   end%function theRes
```

3.15 `theRes()`: wrapper function to zero for equilibria

This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time $t = 1$, and returns the vector of patch-interior time derivatives.

```
15 function f=theRes(u)
16   global patches
17   switch numel(size(patches.x))
18     case 4, pSys = @patchSys1;
19             v=nan(size(patches.x));
20     case 5, pSys = @patchSys2;
21             v=nan(size(patches.x+patches.y));
22     case 6, pSys = @patchSys3;
23             v=nan(size(patches.x+patches.y+patches.z));
24     otherwise error('number of dimensions is somehow wrong')
25   end%switch
26   v(patches.i) = u;
27   f = pSys(1,v(:),patches);
28   f = f(patches.i);
29 end%function theRes
```

3.16 `quasiLogAxes()`: transforms some axes of a plot to quasi-log

This function rescales some coordinates and labels the axes of the given 2D or 3D plot. The original aim was to effectively show the complex spectrum of multiscale systems such as the patch scheme. The eigenvalues are over a wide range of magnitudes, but are signed. So we use a nonlinear asinh transformation of the axes, and then label the axes with reasonable ticks. The nonlinear rescaling is useful in other scenarios also.

```
21 function quasiLogAxes(handle,xScale,yScale,zScale,cScale)
```

Input

- `handle`: handle to your plot to transform, for example, obtained by `handle=plot(...)`
- `xScale` (optional, default inf): if inf, then no transformation is done in the 'x'-coordinate. Otherwise, when `xScale` is not inf, transforms the plot x-coordinates with the asinh() function so that
  - for $|x| \lesssim x_{\text{scale}}$ the x-axis scaling is approximately linear, whereas
  - for $|x| \gtrsim x_{\text{scale}}$ the x-axis scaling is approximately signed-logarithmic.
- `yScale` (optional, default inf): corresponds to `xScale` for the second axis scaling.
- `zScale` (optional, default inf): corresponds to `xScale` for a third axis scaling if it exists.
- `cScale` (optional, default inf): corresponds to `xScale` but for a colormap, and colorbar scaling if one exists.

Output None, just the transformed plot.

Example If invoked with no arguments, then execute an example.

```
58 if nargin==0
59    % generate some data
60    n=99;  fast=(rand(n,1)<0.8);
61    z = -rand(n,1).*(1+1e3*fast)+1i*randn(n,1).*(5+1e2*fast);
62    % plot data and transform axes
63    handle = plot(real(z),imag(z),'.');
64    xlabel('real-part'), ylabel('imag-part')
65    quasiLogAxes(handle,1,10);
66    return
67 end% example
```

Default values for scaling, `inf` denotes no transformation of that axis.

```
76 if nargin<5, cScale=inf; end
77 if nargin<4, zScale=inf; end
78 if nargin<3, yScale=inf; end
79 if nargin<2, xScale=inf; end
```

4 Patches in 2D space

4.1 configPatches2(): configures spatial patches in 2D

Section contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. Section 4.1.1 lists an example of its use.

```
19 function patches = configPatches2(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-04-12';
```

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 4.1.1 for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.
- `Xlim` array/vector giving the rectangular macro-space domain of the computation, namely $[\texttt{Xlim(1)},\texttt{Xlim(2)}]\times[\texttt{Xlim(3)},\texttt{Xlim(4)}]$. If `Xlim` has two elements, then the domain is the square domain of the same interval in both directions.
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is doubly macro-periodic in the 2D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
  - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left/right/bottom/top edges of the leftmost/rightmost/bottommost/topmost patches, respectively.
  - `.bcOffset`, optional one, two or four element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right/top/bottom macroscale boundaries are aligned to the left/right/top/bottom edges of the corresponding extreme patches, but offset by `.bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway between the extreme edge micro-grid points. Similarly for the top and bottom edges.

If `.bcOffset` is a scalar, then apply the same offset to all boundaries. If two elements, then apply the first offset to both x-boundaries, and the second offset to both y-boundaries. If four elements, then apply the first two offsets to the respective x-boundaries, and the last two offsets to the respective y-boundaries.

  - `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x-locations of the centres of the patches—the user is responsible the locations makes sense.
  - `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y-locations of the centres of the patches—the user is responsible the locations makes sense.

- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches (≥ 1) in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.
- `dx` (real—scalar or two element) is usually the sub-patch micro-grid spacing in x and y. If scalar, then use the same `dx` in both directions, otherwise `dx(1:2)` gives the spacing in each of the two directions.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio` (scalar or two element), namely the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the equi-spacing of the patch mid-points—adjusted a little when `nEdge` > 1. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise `nSubP(1:2)` gives the number in each direction. If not using `EdgyInt`, then `nSubP./nEdge` must be odd integer(s) so that there is/are centre-patch lattice lines. So for the defaults of `nEdge` $= 1$ and not `EdgyInt`, then `nSubP` must be odd.
- `'nEdge'`, *optional* (integer—scalar or two element), default=1, the width of edge values set by interpolation at the edge regions of each patch. If two elements, then respectively the width in x, y-directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre cross-patch lines.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- `hetCoeffs`, *optional*, default empty. Supply a 2D or 3D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times n_c$, where n_c is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions (or $n_c = 3$ for the diffusivity tensor). The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1)$-point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of `(nSubP-nEdge)/2` for not `EdgyInt`.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y$ and construct an ensemble of all $m_x \cdot m_y$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x and y directions, respectively, then this coupling cunningly preserves symmetry.
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x, y corresponding to the highest `\nPatch` (if a tie, then chooses the rightmost of x, y). A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`, and `patches.y`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[1]

```
213 if nargout==0, global patches, end
214 patches.version = version;
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.

[1] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl`, only for macro-periodic conditions, are the `ordCC`$\times$ 2-array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified or as derived from `dx`.
- `.x` (6D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (6D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.ratio` 1×2, only for macro-periodic conditions, are the size ratios of every patch.
- `.nEdge` 1×2, is the width of edge values set by interpolation at the edge regions of each patch, in the x, y-directions respectively.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

4.1.1 If no arguments, then execute an example

```
298 if nargin==0
299 disp('With no arguments, simulate example of nonlinear diffusion')
```

The code here shows one way to get started: a user's script may have the following three steps (" $\mapsto$ " denotes function recursion).

1. configPatches2
2. ode23 integrator $\mapsto$ patchSys2 $\mapsto$ user's PDE

3. process results

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, with 5×5 points forming the micro-grid in each patch, and a sub-patch micro-grid spacing of 0.12 (relatively large for visualisation). Roberts et al. (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
global patches
patches = configPatches2(@nonDiffPDE,[-3 3 -2 2] ...
    ,'periodic', [9 7], 0, 0.12, 5 ,'EdgyInt',false);
```

Set an initial condition of a perturbed-Gaussian using auto-replication of the spatial grid.

```
u0 = exp(-patches.x.^2-patches.y.^2);
u0 = u0.*(0.9+0.1*rand(size(u0)));
```

Initiate a plot of the simulation using only the microscale values interior to the patches: optionally set x and y-edges to `nan` to leave the gaps between patches.

```
figure(1), clf, colormap(0.8*hsv)
x = squeeze(patches.x);  y = squeeze(patches.y);
if 1, x([1 end],:) = nan; y([1 end],:) = nan; end
```

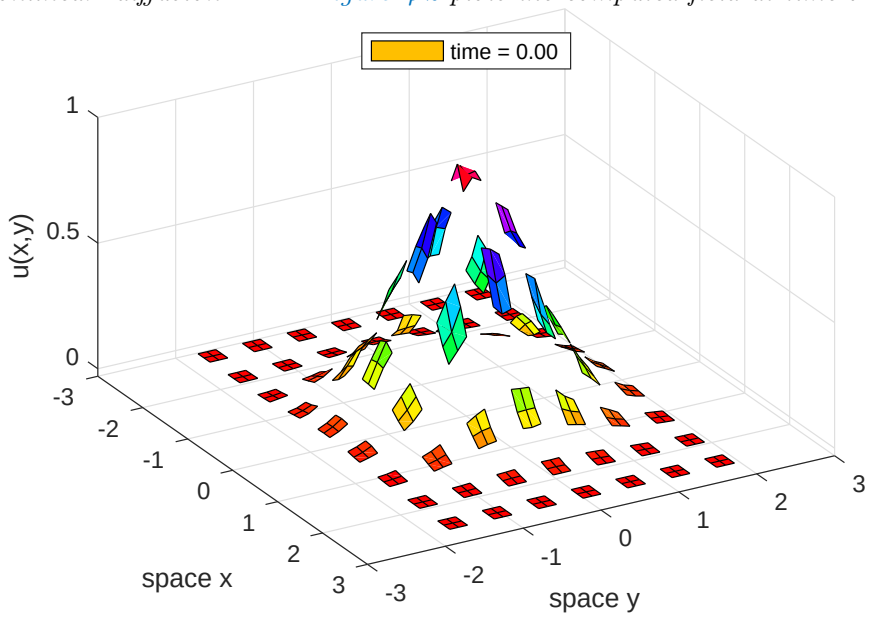
Start by showing the initial conditions of Figure 4.1 while the simulation computes.

```
u = reshape(permute(squeeze(u0) ...
    ,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = mesh(x(:),y(:),u');
axis([-3 3 -3 3 -0.03 1]), view(60,40)
legend('time = 0.00','Location','north')
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
colormap(hsv)
ifOurCf2eps([mfilename 'ic'])
```

Integrate in time to $t = 4$ using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean et al. 2020, Fig. 4). Ask for output at non-uniform times because the diffusion slows.

```
disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys2,linspace(0,2).^2,u0(:));
else % octave version is quite slow for me
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys2,[0 1],u0(:));
end
```

Figure 4.1: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 4.2 plots the computed field at time $t = 3$.



Animate the computed simulation to end with Figure 4.2. Use `patchEdgeInt2` to interpolate patch-edge values.

```
for i = 1:length(ts)
  u = patchEdgeInt2(us(i,:));
  u = reshape(permute(squeeze(u) ...
      ,[1 3 2 4]), [numel(x) numel(y)]);
  set(hsurf,'ZData', u');
  legend(['time = ' num2str(ts(i),'%4.2f')])
  pause(0.1)
end
ifOurCf2eps([mfilename 't3'])
```

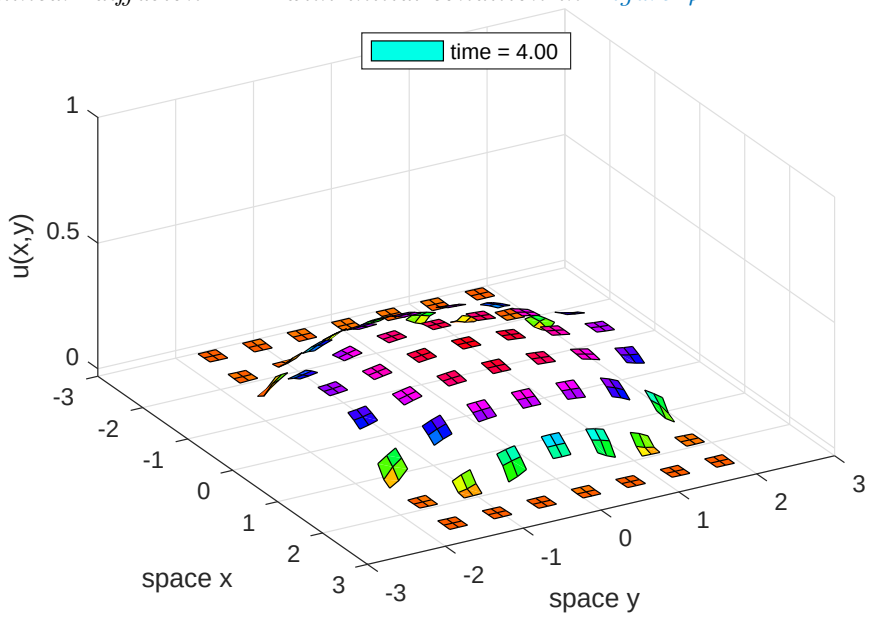
Upon finishing execution of the example, exit this function.

```
return
end%if no arguments
```

Example of nonlinear diffusion PDE inside patches As a microscale discretisation of $u_t = \nabla^2(u^3)$, code $\dot{u}_{ijkl} = \frac{1}{\delta x^2}(u^3_{i+1,j,k,l} - 2u^3_{i,j,k,l} + u^3_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u^3_{i,j+1,k,l} - 2u^3_{i,j,k,l} + u^3_{i,j-1,k,l})$.

```
function ut = nonDiffPDE(t,u,patches)
  if nargin<3, global patches, end
  u = squeeze(u); % reduce to 4D
  dx = diff(patches.x(1:2));  % microgrid spacing
  dy = diff(patches.y(1:2));
  i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior patch points
  ut = nan+u;  % preallocate output array
```

Figure 4.2: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 4.1.




```
20    ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
21                  +diff(u(i,:,:,:).^3,2,2)/dy^2;
22  end
```

4.2 patchSys2(): interface 2D space to time integrators

To simulate in time with 2D spatial patches we often need to interface a users time derivative function with time integration routines such as ode23 or PIRK2. This function provides an interface. Communicate patch-design variables (Section 4.1) either via the global struct patches or via an optional third argument. patches is required for the parallel computing of spmd, or if parameters are to be passed though to the user microscale function.

```
23 function dudt = patchSys2(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

Input

- u is a vector/array of length prod(nSubP)·nVars·nEnsem·prod(nPatch) where there are nVars · nEnsem field values at each of the points in the nSubP(1) × nSubP(2) × nPatch(1) × nPatch(2) grid.
- t is the current time to be passed to the user's time derivative function.
- patches a struct set by configPatches2() with the following information used here.
  - .fun is the name of the user's function fun(t,u,patches,...) that computes the time derivatives on the patchy lattice. The array u has size nSubP(1)×nSubP(2)×nVars×nEsem×nPatch(1)× nPatch(2). Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
  - .x is nSubP(1) × 1 × 1 × 1nPatch(1) × 1 array of the spatial locations x_i of the microscale (i, j)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - .y is similarly $1 \times$ nSubP(2) $\times 1 \times 1 \times$ nPatch(2) array of the spatial locations y_j of the microscale (i, j)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
- varargin, optional, is arbitrary list of parameters to be passed onto the users time-derivative function as specified in configPatches2.

Output

- dudt is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length prod(nSubP) · nVars · nEnsem · prod(nPatch) and the same dimensions as u.

4.3 patchEdgeInt2(): sets 2D patch edge values from 2D macroscale interpolation

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Research (Roberts et al. 2014, Bunder et al. 2021) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder et al. 2020). This function is primarily used by `patchSys2()` but is also useful for user graphics. [2]

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`), or otherwise via the global struct `patches`.

```
29 function u = patchEdgeInt2(u,patches)
30 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP1` · `nSubP2` · `nPatch1` · `nPatch2` multiscale spatial grid on the `nPatch1` · `nPatch2` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is `nSubP1`×1×1×1×`nPatch1`×1 array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times$ `nPatch2` array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j, but may be variable spaced in macroscale index J.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left, right, top and bottom boundaries so interpolation is via divided differences.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in both the x, y-directions—when invoking a periodic domain.
  - `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often

[2] Script `patchEdgeInt2test.m` verifies this code.

preserves symmetry); false, from centre cross-patch values (near original scheme).

- `.nEdge`, two elements, the width of edge values set by interpolation at the x, y-edge regions, respectively, of each patch (default is one for both x, y-edges).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

- `u` is 6D array, `nSubP1` · `nSubP2` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2`, of the fields with edge values set by interpolation.

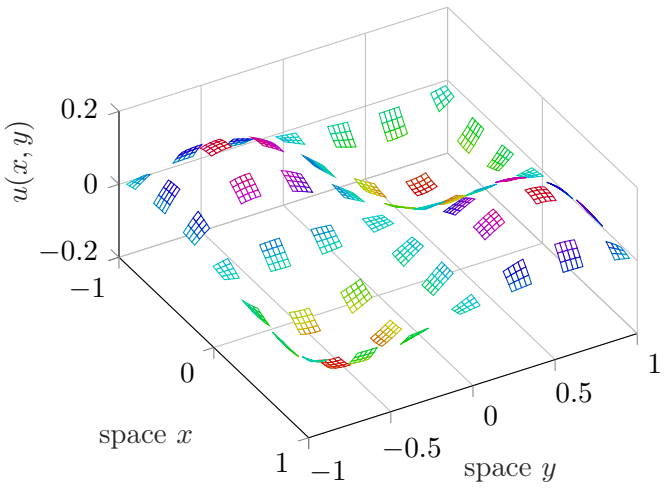


Figure 4.3: Equilibrium of the macroscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.4). The patch size is not small so we can see the patches.

4.4 `monoscaleDiffEquil2`: equilibrium of a 2D monoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x, y)$, see Figure 4.3, to the heterogeneous PDE(inspired by Freese et al.[3] §5.2)

$$u_t = A(x, y)\vec{\nabla}\vec{\nabla}u - f,$$

on domain $[-1, 1]^2$ with Dirichlet BCs, for coefficient pseudo-diffusion matrix

$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \operatorname{sign}(xy) \text{ or } a := \sin(\pi x)\sin(\pi y),$$

and for forcing $f(x, y)$ such that the exact equilibrium is $u = x\big(1-e^{1-|x|}\big)y\big(1-e^{1-|y|}\big)$. But for simplicity, let's obtain $u = x(1 - x^2)y(1 - y^2)$ for which we code f later—as determined by this Reduce algebra code.

```
on gcd; factor sin;
u:=x*(1-x^2)*y*(1-y^2);
a:=sin(pi*x)*sin(pi*y);
f:=2*df(u,x,x)+2*a*df(u,x,y)+2*df(u,y,y);
```

Clear, and initiate globals.

```
57 clear all
58 global patches
59 %global OurCf2eps, OurCf2eps=true %option to save plot
```

Patch configuration Initially use 7×7 patches in the square $(-1, 1)^2$. For continuous forcing we may have small patches of any reasonable microgrid spacing—here the microgrid error dominates.

[3] http://arxiv.org/abs/2211.13731

```
nPatch = 7
nSubP = 5
dx = 0.03
```

Specify some order of interpolation.

```
configPatches2(@monoscaleDiffForce2,[-1 1 -1 1],'equispace' ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true );
```

Compute the time-constant coefficient and time-constant forcing, and store them in struct `patches` for access by the microcode of Section 4.4.1.

```
x=patches.x;  y=patches.y;
patches.A = sin(pi*x).*sin(pi*y);
patches.fu = ...
    +2*patches.A.*(9*x.^2.*y.^2-3*x.^2-3*y.^2+1) ...
    +12*x.*y.*(x.^2+y.^2-2);
```

By construction, the PDE has analytic solution

```
uAnal = x.*(1-x.^2).*y.*(1-y.^2);
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (using `optimoptions` to omit its trace information), and give magnitudes.

```
tic;
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
normResidual = norm(theRes(uSoln))
normSoln = norm(uSoln)
normError = norm(uSoln-uAnal(patches.i))
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
```

```
149  y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
150  u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with boundary-values omitted as already `NaN` by not bothering to set them.

```
157  mesh(x(:),y(:),u'); view(60,55)
158  xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
159  ifOurCf2tex(mfilename)%optionally save
```

4.4.1 `monoscaleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
176  function ut = monoscaleDiffForce2(t,u,patches)
177    dx = diff(patches.x(2:3));  % x space step
178    dy = diff(patches.y(2:3));  % y space step
179    i = 2:size(u,1)-1; % x interior points in a patch
180    j = 2:size(u,2)-1; % y interior points in a patch
181    ut = nan+u;         % preallocate output array
```

Set Dirichlet boundary value of zero around the square domain.

```
188    u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
189    u(end,:,:,:,end,:)=0; % right edge of right patches
190    u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
191    u(:,end,:,:,:,end)=0; % top edge of top patches
```

Or code some function variation around the boundary, such as a function of y on the left boundary, and a (constant) function of x at the top boundary.

```
199    if 0
200      u(1,:,:,:,1,:)=(1+patches.y)/2; % left edge of left patches
201      u(:,end,:,:,:,end)=1; % top edge of top patches
202    end%if
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
210    ut(i,j,:) ...
211    = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
212     +2*patches.A(i,j,:).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
213       -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
214     -patches.fu(i,j,:);
215  end%function monoscaleDiffForce2
```

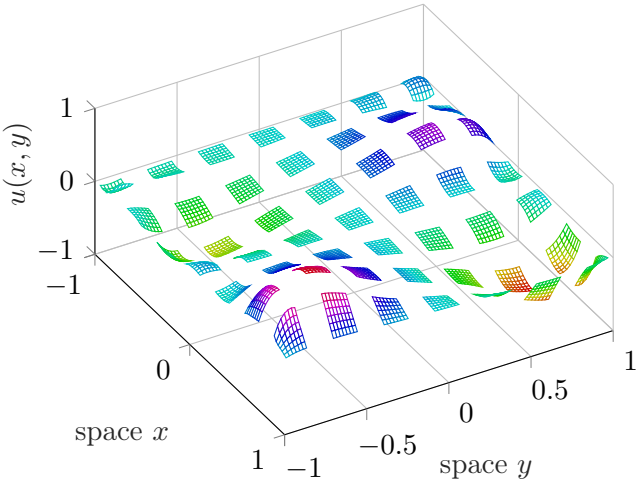


Figure 4.4: Equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.5). The patch size is not small so we can see the patches and the sub-patch grid. The solution $u(x,y)$ is boringly smooth.

4.5 `twoscaleDiffEquil2`: equilibrium of a 2D twoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x,y)$ to the heterogeneous PDE (inspired by Freese et al.[4] §5.3.1)

$$u_t = A(x,y)\vec{\nabla}\vec{\nabla}u - f,$$

on domain $[-1,1]^2$ with Dirichlet BCs, for coefficient 'diffusion' matrix, varying with period 2ϵ on the microscale $\epsilon = 2^{-7}$, of

$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \sin(\pi x/\epsilon)\sin(\pi y/\epsilon),$$

and for forcing $f := (x + \cos 3\pi x)y^3$.

Clear, and initiate globals.

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
```

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Set the phase of the heterogeneity so that each patch centre is a point of symmetry of the diffusivity. Then `configPatches2` replicates the heterogeneity to fill each patch.

```
mPeriod = 6
z = (0.5:mPeriod)'/mPeriod;
A = sin(2*pi*z).*sin(2*pi*z');
```

Set the periodicity, via ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
epsilon = 2^(-6) % 4 or 5 to see the patches
dx = (2*epsilon)/mPeriod
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
```

[4] http://arxiv.org/abs/2211.13731

Patch configuration Say use 7×7 patches in $(-1, 1)^2$, fourth order interpolation, and either 'equispace' or 'chebyshev':

```
nPatch = 7
configPatches2(@twoscaleDiffForce2,[-1 1],'equispace' ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',A );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 4.6.1.

```
x = patches.x;  y = patches.y;
patches.fu = 100*(x+cos(3*pi*x)).*y.^3;
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 3.15), and give magnitudes.

```
tic;
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solveTime = toc
normResidual = norm(theRes(uSoln))
normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
```

Draw the patch solution surface, with boundary-values omitted as already `NaN` by not bothering to set them.

```
mesh(x(:),y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2tex(mfilename)%optionally save
```

4.5.1 `twoscaleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = twoscaleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

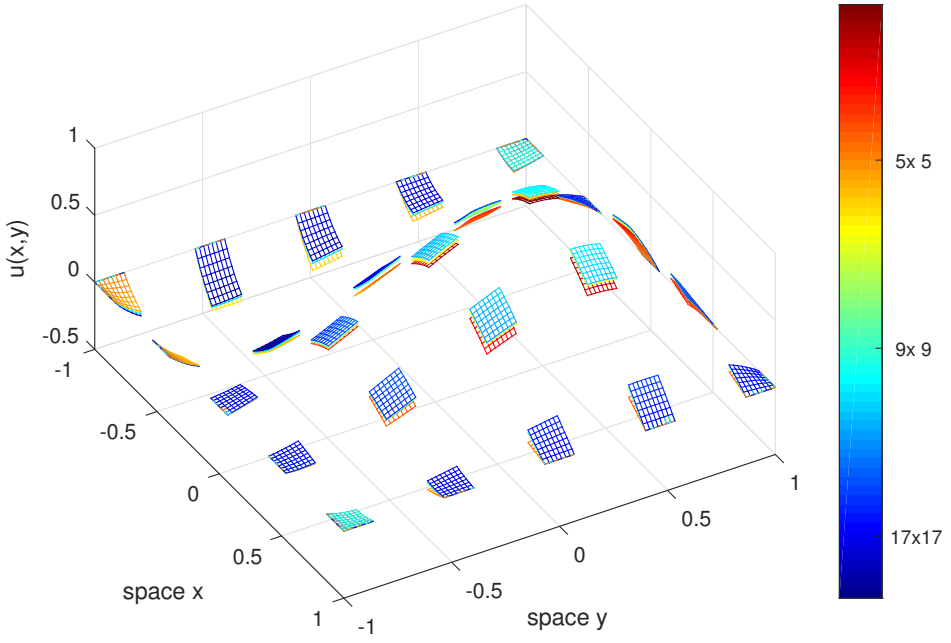
Set Dirichlet boundary value of zero around the square domain.

```
  u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
  u(end,:,:,:,end,:)=0; % right edge of right patches
  u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
  u(:,end,:,:,:,end)=0; % top edge of top patches
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(i,j,:) ...
  = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
   +2*patches.cs(i,j).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
     -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
   -patches.fu(i,j,:);
end%function twoscaleDiffForce2
```

Figure 4.5: For various numbers of patches as indicated on the colorbar, plot the equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.6). We only compare solutions only in these 25 common patches.



4.6 `twoscaleDiffEquil2Errs`: errors in equilibria of a 2D twoscale heterogeneous diffusion via small patches

Here we find the steady state $u(x, y)$ to the heterogeneous PDE (inspired by Freese et al.[5] §5.3.1)

$$u_t = A(x, y)\vec{\nabla}\vec{\nabla}u + f,$$

on domain $[-1, 1]^2$ with Dirichlet BCs, for coefficient 'diffusion' matrix, varying with some microscale period ϵ (here $\epsilon \approx 0.24, 0.12, 0.06, 0.03$), of

$$A := \begin{bmatrix} 2 & a \\ a & 2 \end{bmatrix} \quad \text{with } a := \sin(\pi x/\epsilon)\sin(\pi y/\epsilon),$$

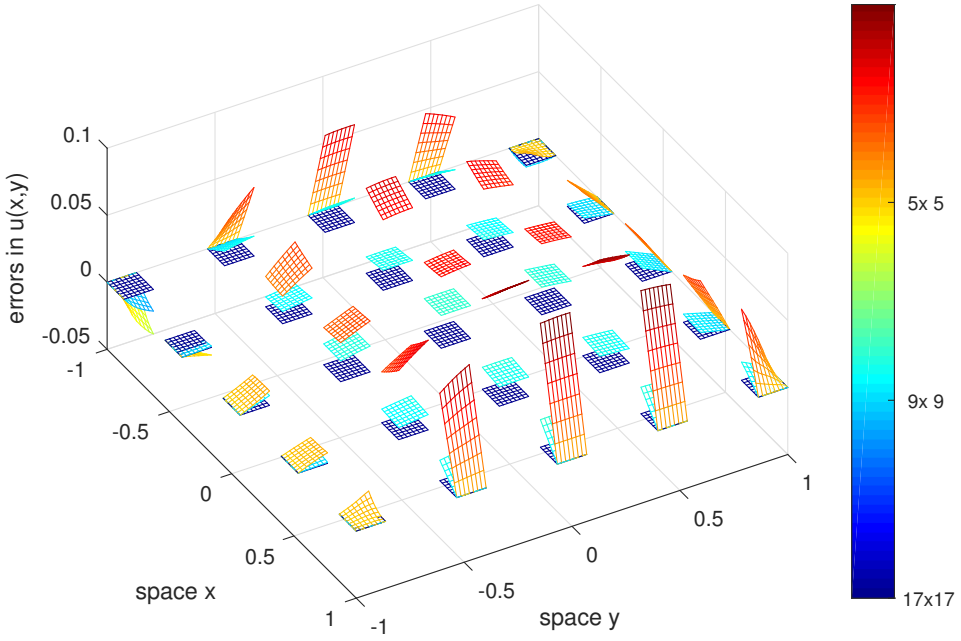
and for forcing $f := 10(x + y + \cos \pi x)$ (for which the solution has magnitude up to one).[6]

Here we explore the errors for increasing number N of patches (in both directions). Find mean-abs errors to be the following (for different orders of

[5] http://arxiv.org/abs/2211.13731

[6] Freese et al. had forcing $f := (x + \cos 3\pi x)y^3$, but here we want smoother forcing so we get meaningful results in a minute or two computation.[7] For the same reason we do not invoke their smaller $\epsilon \approx 0.01$.

Figure 4.6: For various numbers of patches as indicated on the colorbar, plot the equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.6). We only compare solutions only in these 25 common patches.



interpolation and patch distribution):

N	5	9	17	33
equispace, 2nd-order	6E−2	3E−2	1E−2	3E−3
equispace, 4th-order	3E−2	8E−3	7E−4	7E−5
chebyshev, 4th-order	1E−2	2E−2	6E−3	2E−3
usergiven, 4th-order	1E−2	2E−2	4E−3	n/a
equispace, 6th-order	3E−2	1E−3	1E−4	2E−5

Script start Clear, and initiate global patches. Choose the type of patch distribution to be either 'equispace', 'chebyshev', or 'usergiven'. Also set order of interpolation (fourth-order is good start).

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
switch 1
    case 1, Dom.type = 'equispace'
    case 2, Dom.type = 'chebyshev'
    case 3, Dom.type = 'usergiven'
end% switch
ordInt = 4
```

First configure the patch system Establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Then `configPatches2` replicates the heterogeneity as needed to fill each patch.

```
mPeriod = 6
z = (0.5:mPeriod)'/mPeriod;
A = sin(2*pi*z).*sin(2*pi*z');
```

To use a hierarchy of patches with `nPatch` of 5, 9, 17, ..., we need up to N patches plus one `dx` to fit into the domain interval. Cater for up to some full-domain simulation—can compute `log2Nmax` $= 5$ ($\epsilon = 0.06$) within minutes:

```
log2Nmax = 4 % >2 up to 6 OKish
nPatchMax=2^log2Nmax+1
```

Set the periodicity ϵ, and other microscale parameters.

```
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
epsilon = 2/(nPatchMax*nPeriodsPatch+1/mPeriod)
dx = epsilon/mPeriod
```

For various numbers of patches Choose five patches to be the coarsest number of patches. Define variables to store common results for the solutions from differing patches. Assign `Ps` to be the indices of the common patches: for equispace set to the five common patches, but for 'chebyshev' the only common ones are the three centre and boundary-adjacent patches.

```
us=[]; xs=[]; ys=[]; nPs=[];
for log2N=log2Nmax:-1:2
    if log2N==log2Nmax
         Ps=linspace(1,nPatchMax ...
           ,5-2*all(Dom.type=='chebyshev'))
    else Ps=(Ps+1)/2
    end
```

Set the number of patches in $(-1, 1)$:

```
    nPatch = 2^log2N+1
```

In the case of 'usergiven', we set the standard Chebyshev distribution of the patch-centres, which involves overlapping of patches near the boundaries! (instead of the coded chebyshev which has boundary layers of abutting patches, and non-overlapping Chebyshev between the boundary layers).

```
    if all(Dom.type=='usergiven')
      halfWidth = dx*(nSubP-1)/2;
      X1 = -1+halfWidth;  X2 = 1-halfWidth;
      Dom.X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));
      Dom.Y = Dom.X;
    end
```

Configure the patches:

```
    configPatches2(@twoscaleDiffForce2,[-1 1],Dom,nPatch  ...
        ,ordInt ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',A );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 4.6.1.

```
179     if 1
180       patches.fu = 10*(patches.x+cos(pi*patches.x)+patches.y);
181     else patches.fu = 8+0*patches.x+0*patches.y;
182     end
```

Solve for steady state Set initial guess of either zero or a subsample of the previous, next-finer, solution. `NaN` indicates patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
193     if log2N==log2Nmax
194          u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
195     else u0 = u0(:,:,:,:,1:2:end,1:2:end);
196     end
197     u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
198     patches.i = find(~isnan(u0));
199     nVariables = numel(patches.i)
```

First try to solve via iterative solver `bicgstab`, via the generic patch system wrapper `theRes` (Section 3.15).

```
207     tic;
208     maxIt = ceil(nVariables/10);
209     rhsb = theRes( zeros(size(patches.i)) );
210     [uSoln,flag] = bicgstab(@(u) rhsb-theRes(u),rhsb ...
211                     ,1e-9,maxIt,[],[],u0(patches.i));
212     bicgTime = toc
```

However, the above often fails (and `fsolve` sometimes takes too long here), so then try a preconditioned version of `bicgstab`. The preconditioner is derived from the Jacobian which is expensive to find (four minutes for $N = 33$, one hour for $N = 65$), but we do so as follows.

```
222     if flag>0, disp('**** bicg failed, trying ILU preconditioner')
223         disp(['Computing Jacobian: wait roughly ' ...
224               num2str(nPatch^4/4500,2) ' secs'])
225         tic
226         Jac=sparse(nVariables,nVariables);
227         for j=1:nVariables
228             Jac(:,j)=sparse( rhsb-theRes((1:nVariables)'==j) );
229         end
230         formJacTime=toc
```

Compute an incomplete LU-factorization, and use it as preconditioner to `bicgstab`.

```
237         tic
238         [L,U] = ilu(Jac,struct('type','ilutp','droptol',1e-4));
239         LUfillFactor = (nnz(L)+nnz(U))/nnz(Jac)
```

```
240         [uSoln,flag] = bicgstab(@(u) rhsb-theRes(u),rhsb ...
241                     ,1e-9,maxIt,L,U,u0(patches.i));
242         precondSolveTime=toc
243         assert(flag==0,'preconditioner fails bicgstab. Lower droptol?')
244     end%if flag
```

Store the solution into the patches, and give magnitudes—Inf norm is max(abs()).

```
251     normResidual = norm(theRes(uSoln),Inf)
252     normSoln = norm(uSoln,Inf)
253     u0(patches.i) = uSoln;
254     u0 = patchEdgeInt2(u0);
255     u0( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
256     u0(end,:,:,:,end,:)=0; % right edge of right patches
257     u0(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
258     u0(:,end,:,:,:,end)=0; % top edge of top patches
259     assert(normResidual<1e-5,'poor--bad solution found')
```

Concatenate the solution on common patches into stores.

```
265     us=cat(5,us,squeeze(u0(:,:,:,:,Ps,Ps)));
266     xs=cat(3,xs,squeeze(patches.x(:,:,:,:,Ps,:)));
267     ys=cat(3,ys,squeeze(patches.y(:,:,:,:,:,Ps)));
268     nPs = [nPs;nPatch];
```

End loop. Check micro-grids are aligned, then compute errors compared to the full-domain solution (or the highest resolution solution for the case of 'usergiven').

```
277 end%for log2N
278 assert(max(abs(reshape(diff(xs,1,3),[],1)))<1e-12,'x-coord failure')
279 assert(max(abs(reshape(diff(ys,1,3),[],1)))<1e-12,'y-coord failure')
280 errs = us-us(:,:,:,:,1);
281 meanAbsErrs = mean(abs(reshape(errs,[],size(us,5))))
282 ratioErrs = meanAbsErrs(2:end)./meanAbsErrs(1:end-1)
```

Plot solution in common patches First reshape arrays to suit 2D space surface plots, inserting nans to separate patches.

```
294 x = xs(:,:,1); y = ys(:,:,1); u=us;
295 x(end+1,:)=nan;   y(end+1,:)=nan;
296 u(end+1,:,:)=nan; u(:,end+1,:)=nan;
297 u = reshape(permute(u,[1 3 2 4 5]),numel(x),numel(y),[]);
```

Plot the patch solution surfaces, with colour offset between surfaces (best if u-field has a range of one): blues are the full-domain solution, reds the coarsest patches.

```
305 figure(1), clf, colormap(jet)
306 for p=1:size(u,3)
307      mesh(x(:),y(:),u(:,:,p)',p+u(:,:,p)');
308      hold on;
```

```
end, hold off
view(60,55)
colorbar('Ticks',1:size(u,3) ...
    ,'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2eps([mfilename 'us'])%optionally save
```

Plot error surfaces Plot the error surfaces, with colour offset between surfaces (best if u-field has a range of one): dark blue is the full-domain zero error, reds the coarsest patches.

```
err=u(:,:,1)-u;
maxAbsErr=max(abs(err(:)));
figure(2), clf, colormap(jet)
for p=1:size(u,3)
     mesh(x(:),y(:),err(:,:,p)',p+err(:,:,p)'/maxAbsErr);
     hold on;
end, hold off
view(60,55)
colorbar('Ticks',1:size(u,3) ...
    ,'TickLabels',[num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
xlabel('space $x$'), ylabel('space $y$')
zlabel('errors in $u(x,y)$')
ifOurCf2eps(mfilename)%optionally save
```

4.6.1 `twoscaleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = twoscaleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

Set Dirichlet boundary value of zero around the square domain.

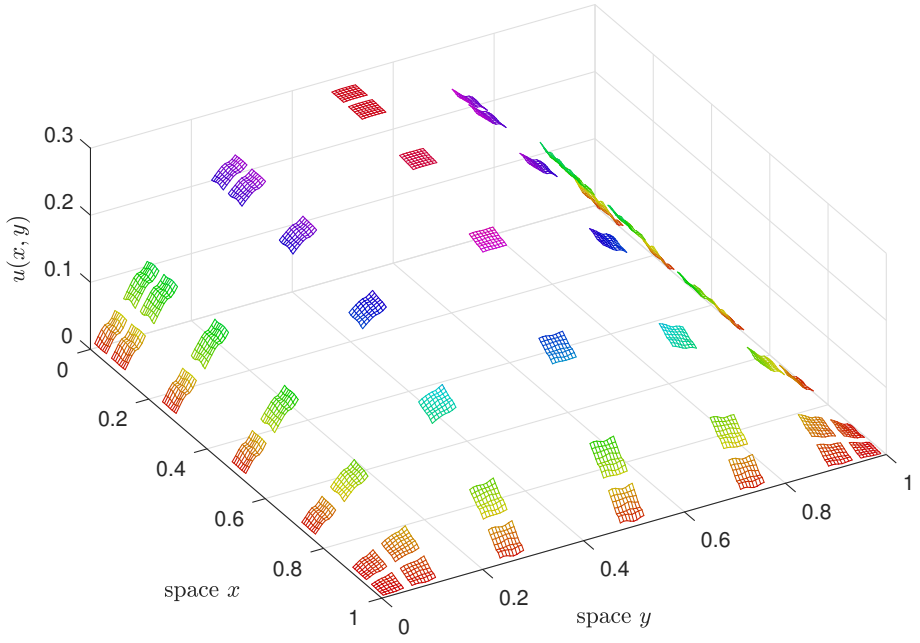
```
  u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
  u(end,:,:,:,end,:)=0; % right edge of right patches
  u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
  u(:,end,:,:,:,end)=0; % top edge of top patches
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(i,j,:) ...
  = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
   +2*patches.cs(i,j).*( u(i+1,j+1,:) -u(i-1,j+1,:) ...
```

```
383       -u(i+1,j-1,:) +u(i-1,j-1,:) )/(4*dx*dy) ...
384     +patches.fu(i,j,:);
385  end%function twoscaleDiffForce2
```

Figure 4.7: Equilibrium of the macroscale diffusion problem of Abdulle with boundary conditions of Dirichlet zero-value except for $x = 0$ which is Neumann (Section 4.7). Here the patches have a Chebyshev-like spatial distribution. The patch size is chosen large enough to see within.



4.7 `abdulleDiffEquil2`: equilibrium of a 2D multiscale heterogeneous diffusion via small patches

Here we find the steady state $u(x, y)$ to the heterogeneous PDE (inspired by Abdulle et al. 2020, §5.1)

$$u_t = \vec{\nabla} \cdot [a(x, y)\vec{\nabla}u] + 10,$$

on square domain $[0, 1]^2$ with zero-Dirichlet BCs, for coefficient 'diffusion' matrix, varying with period ϵ of (their (45))

$$a := \frac{2 + 1.8 \sin 2\pi x/\epsilon}{2 + 1.8 \cos 2\pi y/\epsilon} + \frac{2 + \sin 2\pi y/\epsilon}{2 + 1.8 \cos 2\pi x/\epsilon}.$$

Figure 4.7 shows solutions have some nice microscale wiggles reflecting the heterogeneity.

Clear, and initiate globals.

```
38 clear all
39 global patches
40 %global OurCf2eps, OurCf2eps=true %option to save plot
```

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial micro-grid lattice. Then `configPatches2` replicates the heterogeneity to fill each patch. (These diffusion coefficients should really recognise the half-grid-point shifts, but let's not bother.)

```
mPeriod = 6
x = (0.5:mPeriod)'/mPeriod;  y=x';
a = (2+1.8*sin(2*pi*x))./(2+1.8*sin(2*pi*y)) ...
   +(2+    sin(2*pi*y))./(2+1.8*sin(2*pi*x));
diffusivityRange = [min(a(:)) max(a(:))]
```

Set the periodicity ϵ, here big enough so we can see the patches, and other microscale parameters.

```
epsilon = 0.04
dx = epsilon/mPeriod
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % when edgy int
```

Patch configuration Choose either Dirichlet (default) or Neumann on the left boundary in coordination with micro-code in Section 4.7.1

```
Dom.bcOffset = zeros(2);
if 1, Dom.bcOffset(1)=0.5; end% left Neumann
```

Say use 7×7 patches in $(0,1)^2$, fourth order interpolation, and either 'equispace' or 'chebyshev':

```
nPatch = 7
Dom.type='chebyshev';
configPatches2(@abdulleDiffForce2,[0 1],Dom ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',a );
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, and the number of unknowns is then its length.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 3.15), and give magnitudes.

```
tic;
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
normResidual = norm(theRes(uSoln))
normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
figure(1), clf, colormap(0.8*hsv)
patches.x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
patches.y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
u = reshape(permute(squeeze(u0),[1 3 2 4]) ...
    , [numel(patches.x) numel(patches.y)]);
```

Draw the patch solution surface, with boundary-values omitted as already `NaN` by not bothering to set them.

```
mesh(patches.x(:),patches.y(:),u'); view(60,55)
xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
ifOurCf2eps(mfilename) %optionally save plot
```

4.7.1 `abdulleDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
function ut = abdulleDiffForce2(t,u,patches)
  dx = diff(patches.x(2:3));  % x space step
  dy = diff(patches.y(2:3));  % y space step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  ut = nan+u;         % preallocate output array
```

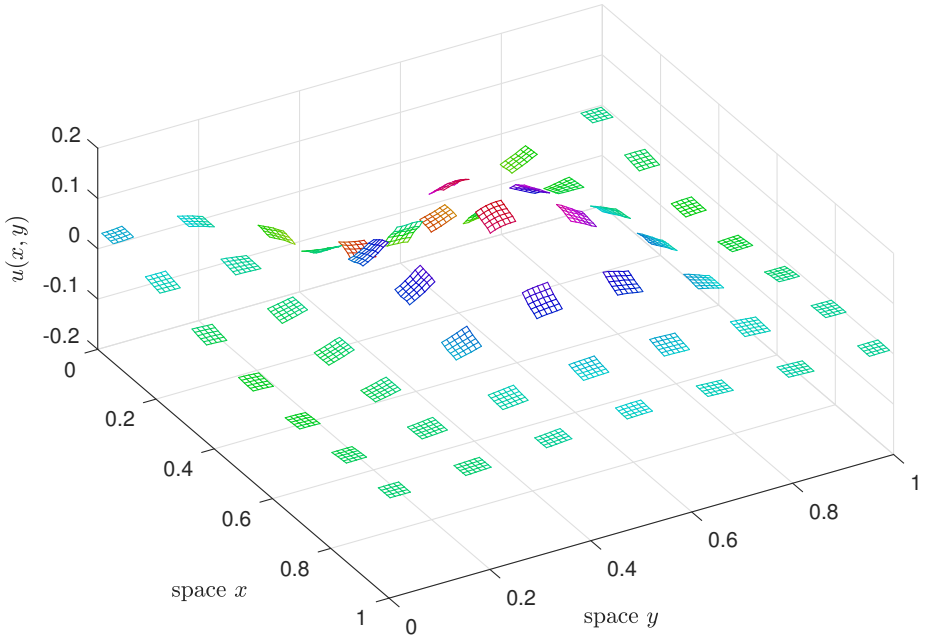
Set Dirichlet boundary value of zero around the square domain, but also cater for zero Neumann condition on the left boundary.

```
  u( 1 ,:,:,:, 1 ,:)=0; % left edge of left patches
  u(end,:,:,:,end,:)=0; % right edge of right patches
  u(:, 1 ,:,:,:, 1 )=0; % bottom edge of bottom patches
  u(:,end,:,:,:,end)=0; % top edge of top patches
  if 1, u(1,:,:,:,1,:)=u(2,:,:,:,1,:); end% left Neumann
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
  ut(i,j,:) = diff(patches.cs(:,j).*diff(u(:,j,:)))/dx^2 ...
      + diff(patches.cs(i,:).*diff(u(i,:,:),1,2),1,2)/dy^2 ...
      + 10;
end%function abdulleDiffForce2
```

Figure 4.8: Equilibrium of the macroscale diffusion problem of Bonizzoni et al. with Neumann boundary conditions of zero (Section 4.8). Here the patches have a equispaced spatial distribution. The microscale periodicity, and hence the patch size, is chosen large enough to see within.



4.8 `randAdvecDiffEquil2`: equilibrium of a 2D random heterogeneous advection-diffusion via small patches

Here we find the steady state $u(x,y)$ of the heterogeneous PDE (inspired by Bonizzoni et al.[8] §6.2)

$$u_t = \mu_1 \nabla^2 u - (\cos\mu_2, \sin\mu_2)\cdot\vec{\nabla}u - u + f\,,$$

on domain $[0,1]^2$ with Neumann boundary conditions, for microscale random pseudo-diffusion and pseudo-advection coefficients, $\mu_1(x,y) \in [0.01, 0.1]$[9] and $\mu_2(x,y) \in [0, 2\pi)$, and for forcing

$$f(x,y) := \exp\left[-\frac{(x-\mu_3)^2 + (y-\mu_4)^2}{\mu_5^2}\right],$$

smoothly varying in space for fixed $\mu_3, \mu_4 \in [0.25, 0.75]$ and $\mu_5 \in [0.1, 0.25]$. The above system is dominantly diffusive for lengths scales $\ell < 0.01 = \min\mu_1$. Due to the randomness, we get different solutions each execution of this code. Figure 4.8 plots one example. A physical interpretation of the solution field is confounded because the problem is pseudo-advection-diffusion due to the varying coefficients being outside the $\vec{\nabla}$ operator.

Clear, and initiate globals.

```
clear all
global patches
%global OurCf2eps, OurCf2eps=true %option to save plot
```

[8] http://arxiv.org/abs/2211.15221

[9] More interesting microscale structure arises here for μ_1 a factor of three smaller.

First establish the microscale heterogeneity has micro-period `mPeriod` on the spatial lattice. Then `configPatches2` replicates the heterogeneity to fill each patch.

```
mPeriod = 4
mu1 = 0.01*10.^rand(mPeriod)
mu2 = 2*pi*rand(mPeriod)
cs = cat(3,mu1,cos(mu2),sin(mu2));
meanDiffAdvec=squeeze(mean(mean(cs)))
```

Set the periodicity ϵ, here big enough so we can see the patches, and other microscale parameters.

```
epsilon = 0.04
dx = epsilon/mPeriod
nPeriodsPatch = 1 % any integer
nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
```

Patch configuration Say use 7×7 patches in $(0,1)^2$, fourth order interpolation, either 'equispace' or 'chebyshev', and the offset for Neumann boundary conditions:

```
nPatch = 7
Dom.type= 'equispace';
Dom.bcOffset = 0.5;
configPatches2(@randAdvecDiffForce2,[0 1],Dom ...
    ,nPatch ,4 ,dx ,nSubP ,'EdgyInt',true ,'hetCoeffs',cs );
```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of Section 4.8.1.

```
mu = [ 0.25+0.5*rand(1,2) 0.1+0.15*rand ]
patches.fu = exp(-((patches.x-mu(1)).^2 ...
                  +(patches.y-mu(2)).^2)/mu(3)^2);
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, store in global patches for access by `theRes`, and the number of unknowns is then its number of elements.

```
u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
u0([1 end],:,:) = nan;  u0(:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (and using `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 3.15).

```
tic;
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solnTime = toc
```

```
134 normResidual = norm(theRes(uSoln))
135 normSoln = norm(uSoln)
```

Store the solution vector into the patches, and interpolate, but have not bothered to set boundary values so they stay NaN from the interpolation.

```
143 u0(patches.i) = uSoln;
144 u0 = patchEdgeInt2(u0);
```

Draw solution profile Separate patches with NaNs, then reshape arrays to suit 2D space surface plots.

```
155 figure(1), clf, colormap(0.8*hsv)
156 patches.x(end+1,:,:)=nan;  u0(end+1,:,:)=nan;
157 patches.y(:,end+1,:)=nan;  u0(:,end+1,:)=nan;
158 u = reshape(permute(squeeze(u0),[1 3 2 4]) ...
159     , [numel(patches.x) numel(patches.y)]);
```

Draw the patch solution surface, with boundary-values omitted as already `NaN` by not bothering to set them.

```
166 mesh(patches.x(:),patches.y(:),u'); view(60,55)
167 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
168 ifOurCf2eps(mfilename) %optionally save plot
```

4.8.1 `randAdvecDiffForce2()`: microscale discretisation inside patches of forced diffusion PDE

This function codes the lattice heterogeneous diffusion of the PDE inside the patches. For 6D input arrays `u`, `x`, and `y`, computes the time derivative at each point in the interior of a patch, output in `ut`.

```
187 function ut = randAdvecDiffForce2(t,u,patches)
188   dx = diff(patches.x(2:3));  % x space step
189   dy = diff(patches.y(2:3));  % y space step
190   i = 2:size(u,1)-1; % x interior points in a patch
191   j = 2:size(u,2)-1; % y interior points in a patch
192   ut = nan+u;         % preallocate output array
```

Set Neumann boundary condition of zero derivative around the square domain: that is, the edge value equals the next-to-edge value.

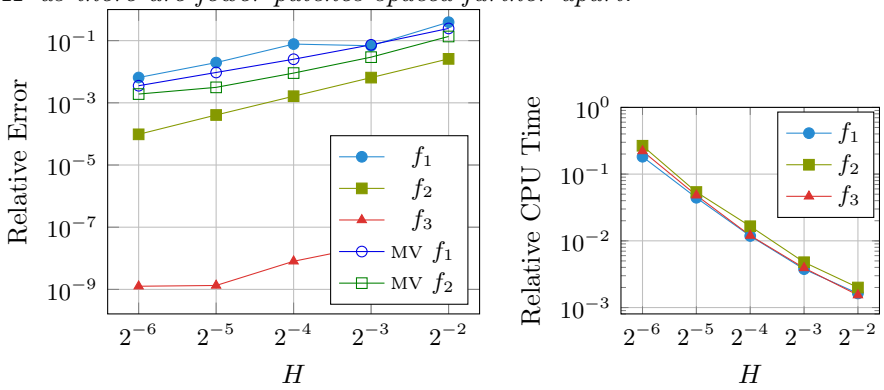
```
200   u( 1 ,:,:,:, 1 ,:)=u(  2  ,:,:,:, 1 ,:); % left edge of left patches
201   u(end,:,:,:,end,:)=u(end-1,:,:,:,end,:); % right edge of right patches
202   u(:, 1 ,:,:,:, 1 )=u(:,  2  ,:,:,:, 1 ); % bottom edge of bottom patches
203   u(:,end,:,:,:,end)=u(:,end-1,:,:,:,end); % top edge of top patches
```

Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one `,:`.

```
211   ut(i,j,:) ...
212   = patches.cs(i,j,1).*(diff(u(:,j,:),2,1)/dx^2 ...
213                         +diff(u(i,:,:),2,2)/dy^2)...
214    -patches.cs(i,j,2).*(u(i+1,j,:)-u(i-1,j,:))/(2*dx) ...
```

```
215     -patches.cs(i,j,3).*(u(i,j+1,:)-u(i,j-1,:))/(2*dy) ...
216     -u(i,j,:) +patches.fu(i,j,:);
217   end%function randAdvecDiffForce2
```

Figure 4.9: results for the computational homogenisation of a forced, non-autonomous, 2D wave (Section 4.9). (left) relative RMS *error of the patch scheme, each patch of width* $1/128$*, as a function of patch spacing* H*. The unfilled symbols are those of the energy norm from Maier & Verfürth (2021) (their Figure 5.1). (right) the relative compute time decreases very quickly in* H *as there are fewer patches spaced further apart.*



4.9 `homoWaveEdgy2`: computational homogenisation of a forced, non-autonomous, 2D wave via simulation on small patches

This section extends to 2D waves, in a microscale heterogeneous media, the 2D diffusion code discussed in **??**. It favourably compares to the examples of Maier & Verfürth (2021).

Figure 4.9 summarises the results here. The left (larger) graph shows the error in the patch scheme decreasing with decreasing patch spacing H (increasing number of patches). Forcing f_1 and f_2 are as specified by §5.1 of Maier & Verfürth (2021), whereas f_3 here is f in their §5.2. For the case of forcing f_1 which is discontinuous in space (at $x = 0.4$), the errors are similar to that of Maier & Verfürth (2021)—compare the filled with unfilled circles. For the case of forcing f_2 which is continuous in the spatial domain, except for a second derivative discontinuity in its odd-periodic extension, the errors of the patch scheme are an order of magnitude better that that of Maier & Verfürth (2021)—compare the filled with unfilled squares. For the case of forcing f_3 which is smooth in the domain and in its odd-periodic extension, the patch scheme errors, roughly 10^{-8}, are at the tolerance of the time integration. Two caveats in a comparison with Maier & Verfürth (2021) are the slightly different norms used, and that they also address errors in the time integration, whereas here we use a standard adaptive integrator in order to focus purely on the spatial errors of the patch scheme.

Now let's code the simulation of the forced, non-autonomous, 2D wave. Maier & Verfürth (2021) have Dirichlet BCs of zero around the unit square, so replicate here by the odd periodic extension to the spatial domain $[-1, 1]^2$. In their §5.1, their microscale mesh step is $1/512 = 2^{-9}$. Coding that here results in a compute time of roughly 90 minutes, so here I provide a much coarser case that computes in only a few minutes: change as you please.

```
clear all
dx = 1/128  % 1/512=2^{-9} is the original, but takes 90 mins
```

The heterogeneity is of period four on the microscale lattice, so code a minimal patch size that covers one period.

```
epsilon = 4*dx
nPeriodsPatch = 1
mPeriod = round(epsilon/dx)
nSubP = mPeriod*nPeriodsPatch+2
```

Choose which of three forcing functions to use

```
fn=2
```

Maier & Verfürth (2021) use varying number of macroscale grid steps from 4 to 64 on $[0,1]$ so here on $[-1,1]$ we use double the number patches in each direction. Loop over the number of patches used, starting with the full domain simulation, and then progressively coarsening the macroscale grid of patches.

```
nPatch = 2/epsilon/nPeriodsPatch
for iPat=0:9
if iPat>0, nPatch=nPatch/2, end
if nPatch<8, break, end
```

Set the periodic heterogeneous coefficient, isotropic:

$$a_\epsilon(t,x) = \big[3+\sin(2\pi x/\epsilon)+\sin(2\pi t)\big]\cdot\big[3+\sin(2\pi y/\epsilon)+\sin(2\pi t)\big],$$

which being in product form with two time-dependencies we store as the two spatially varying factors—although to preserve odd symmetry we phase shift the heterogeneity from sines to cosines. It is a user's choice whether to code such spatial dependencies here with `cHetr` or within the time derivative function itself. In this case, I choose to code microscale heterogeneous coefficients here via `cHetr`, and the macroscale variation of f_i in the time derivative function.

Here the period of the heterogeneity is only four microscale lattice points in each direction (which is pretty inaccurate on the microscale, but immaterial as we and Maier & Verfürth (2021) only compare to the coded system on the microscale lattice, not to the PDE). With the following careful choices we ensure all the hierarchy of patch schemes both maintain odd symmetry, and also compute on grid points that are common with the full domain.

```
ratio = (nSubP-2)*dx/(2/nPatch)
Xleft=(1-ratio)/nPatch;
xmid=Xleft+dx*(0:mPeriod-1)';    % half-points
xi = Xleft+dx*(-0.5:mPeriod-1)'; % grid-points
% two components for ax, the x-dirn interactions
cHetr(:,:,1) = (3+cos(2*pi*xmid/epsilon))+0*xi';
cHetr(:,:,2) = 0*xmid+(3+cos(2*pi*xi'/epsilon));
% two components for ay, the y-dirn interactions
cHetr(:,:,3) = (3+cos(2*pi*xi/epsilon))+0*xmid';
cHetr(:,:,4) = 0*xi+(3+cos(2*pi*xmid'/epsilon));
```

Configure patches using spectral interpolation. Quadratic interpolation did not seem significantly different for the case of discontinuous forcing f_1.

```
configPatches2(@heteroWave2,[-1 1 -1 1],nan,nPatch ...
    ,0,ratio,nSubP ,'EdgyInt',true ,'hetCoeffs',cHetr );
```

A check on the spatial geometry.

```
global patches
dxPat=diff(patches.x(1:2));
assert(abs(dx-dxPat)<1e-9,"dx mismatch")
```

Simulate Set the particular forcing function to use, and the zero initial conditions of a simulation.

```
patches.eff=fn;
clear uv0
uv0(:,:,1,1,:,:) = 0*patches.x+0*patches.y;
uv0(:,:,2,1,:,:) = 0*patches.x+0*patches.y;
```

Integrate using standard integrators. Maier & Verfürth (2021) use a scheme with fixed time-step of $\tau = 2^{-7} = 1/128$. Here `ode23` uses variable steps of about 0.0003, and takes 7 s for `nPatch=2*4` (whereas `ode15s` takes 149 s—even for the dissipating case), and takes 287 s for `nPatch=2*32` and roughly 4000 s for full domain `nPatch=2*128`.

```
disp('Now simulate over time')
tic
[ts,us] = ode23(@patchSys2, linspace(0,1,11), uv0(:));
if iPat==0,  odeTime0=toc
else relOdeTime(iPat)=toc/odeTime0
end
```

Compute error compared to full domain simulation Get spatial coordinates of patch-interior points, and reshape to column vectors.

```
i = 2:nSubP-1;
x = squeeze(patches.x(i,:,:,:,:,:));
y = squeeze(patches.y(:,i,:,:,:,:));
x=x(:); y=y(:);
```

At the final time of $t = 1$, get the row vector of data, form into the 6D array via the interpolation to the edges, and reshape patch-interior points to 2D spatial array.

```
uv = squeeze( patchEdgeInt2(us(end,:)));
u = squeeze( uv(i,i,1,:,:) );
u = reshape(permute(u,[1 3 2 4]),[numel(x) numel(y)]);
```

If this is the full domain simulation, then store as the reference solution.

```
if iPat==0
  x0=x; y0=y; u0=u;
```

```
219     rms0=sqrt(mean(u0(:).^2))
220   else
```

Else compute the error compared to the full domain solution. First find the indices of the full domain that match the spatial locations of the patch scheme.

```
228     [i,k] = find(abs(x0-x')<1e-9);
229     assert(length(i)==length(x),'find error in index i')
230     [j,k] = find(abs(y0-y')<1e-9);
231     assert(length(j)==length(y),'find error in index j')
```

The RMS error over the surface is

```
237     errs=u-u0(i,j);
238     relrmserr(iPat)=sqrt(mean(errs(:).^2))/rms0
239     H(iPat)=2/nPatch
240   end%if iPat
```

End the loop over the various number of patches, and return. Further, here not executed, code in the file animates the solution over time, and computes spectrum of the system.

```
250   end%for iPat
251   figure(1), clf
252   loglog(H,relrmserr,'o:'),  grid on
253   xlabel('$H$'), ylabel('relative error')
254   return
```

4.9.1 `heteroWave2()`: heterogeneous Waves

This function codes the lattice heterogeneous waves inside the patches. The forced wave PDE is

$$u_t = v, \quad v_t = \vec{\nabla}(a\vec{\nabla}\cdot u) + f$$

for scalars $a(t,x,y)$ and $f(t,x,y)$ where a has microscale variations. For 6D input arrays `u`, `x`, and `y` (via edge-value interpolation of `patchSys2`, Section 4.2), computes the time derivative at each point in the interior of a patch, output in `ut`. The four 2D arrays of heterogeneous interaction coefficients, c_{ijk}, have previously been stored in `patches.cs` (3D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
26  function ut = heteroWave2(t,u,patches)
27    if nargin<3, global patches, end
```

Microscale space-steps, and interior point indices.

```
33    dx = diff(patches.x(2:3));  % x micro-scale step
34    dy = diff(patches.y(2:3));  % y micro-scale step
35    i = 2:size(u,1)-1; % x interior points in a patch
36    j = 2:size(u,2)-1; % y interior points in a patch
37    assert(max(abs(u(:)))<9999,"u-field exploding")
```

Form coefficients here—odd periodic extension. To avoid slight errors in periodicity (in full domain simulation), first adjust any coordinates crossing $x = \pm 1$ or $y = \pm 1$.

```
x=patches.x; y=patches.y;
l=find(abs(x)>1); x(l)=x(l)-sign(x(l))*2;
l=find(abs(y)>1); y(l)=y(l)-sign(y(l))*2;
```

Then set at this time three possible forcing functions, although only use one depending upon `patches.eff`. Forcing f_1 and f_2 are as specified by §5.1 of Maier & Verfürth (2021), whereas f_3 here is f in their §5.2.

```
f1 = ( (abs(x)>0.4)*(20*t+230*t^2) ...
      +(abs(x)<0.4)*(100*t+2300*t^2) ).*sign(x).*sign(y);
f2 = 20*t*x.*(1-abs(x)).*y.*(1-abs(y)) ...
    +230*t^2*(sign(y).*x.*(1-abs(x))+sign(x).*y.*(1-abs(y)));
f3 = (5*t+50*t^2)*sin(pi*x).*sin(pi*y);
```

Also set the heterogeneous interactions at this time.

```
ax = (patches.cs(:,:,1)+sin(2*pi*t)) ...
   .*(patches.cs(:,:,2)+sin(2*pi*t));
ay = (patches.cs(:,:,3)+sin(2*pi*t)) ...
   .*(patches.cs(:,:,4)+sin(2*pi*t));
```

Reserve storage (using `nan+u` appears quickest), and then assign time derivatives for interior patch values due to the heterogeneous interaction and forcing.

```
  ut = nan+u;  % preallocate output array
  ut(i,j,1,:) = u(i,j,2,:);
  ut(i,j,2,:) ...
  = diff(ax(:,j).*diff(u(:,j,1,:),1),1)/dx^2 ...
   +diff(ay(i,:).*diff(u(i,:,1,:),1,2),1,2)/dy^2 ...
   +(patches.eff==1)*f1(i,j,:,:) ...
   +(patches.eff==2)*f2(i,j,:,:) ...
   +(patches.eff==3)*f3(i,j,:,:) ...
   + 1e-4*(diff(u(:,j,2,:),2,1)/dx^2+diff(u(i,:,2,:),2,2)/dy^2);
end% function
```

In the last line above, the slight damping of 10^{-4} causes microscale modes to decay at rate e^{-28t}, with frequencies 2000–5000, whereas macroscale modes decay with rates roughly 0.0005–0.05 with frequencies 10–100. This slight damping term may correspond to the weak damping of the backward Euler scheme adopted by Maier & Verfürth (2021) for time integration.

4.10 SwiftHohenberg2dPattern: patterns of the Swift–Hohenberg PDE in 2D on patches

Section contents

Figures 4.10 to 4.15 show an example simulation in time generated by the patch scheme applied to the patterns arising from the 2D Swift–Hohenberg PDE.

Consider a lattice of values $u_i(t)$, with lattice spacing dx, and governed by a microscale centred discretisation of the Swift–Hohenberg PDE

$$\partial_t u = -(1 + \nabla^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3, \tag{4.1}$$

with various boundary conditions at $x, y = 0, L$. For Ra just above critical, say $\mathrm{Ra} = 0.1$, the system rapidly evolves to spatial quasi-periodic solutions with period ≈ 0.24 when wavenumber parameter $k_0 = 26$. These spatial oscillations are here resolved on a micro-grid of spacing 0.042. On medium times these spatial oscillations grow to near equilibrium amplitude of $\sqrt{\mathrm{Ra}}$, and over very long times the phases of the oscillations evolve in space to adapt to the boundaries.

Set the desired microscale periodicity, and correspondingly choose random microscale diffusion coefficients (with subscripts shifted by a half).

```
clear all
cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
%global OurCf2eps, OurCf2eps=true %optional to save plots
Ra = 0.2 % Ra>0 leads to patterns
nGapFac = 2
waveLength = 0.5/nGapFac
nPtsPeriod = 6
dx = waveLength/nPtsPeriod
k0 = 2.1*pi/waveLength
```

The above factor 2.1 is close to $3/\sqrt{2} = 2.1213$ for which $(\pm 1, \pm 2)$ modes have same linear growth-rate as $(\pm 2, 0)$ modes.

Establish global data struct `patches` for the Swift–Hohenberg PDE on some square domain. For simplicity, use five patches in each direction. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Set `bcOffset` for different boundary conditions around the square domain.

```
nPatch = 5
nSubP = 2*nPtsPeriod+4
Len = nPatch;
```

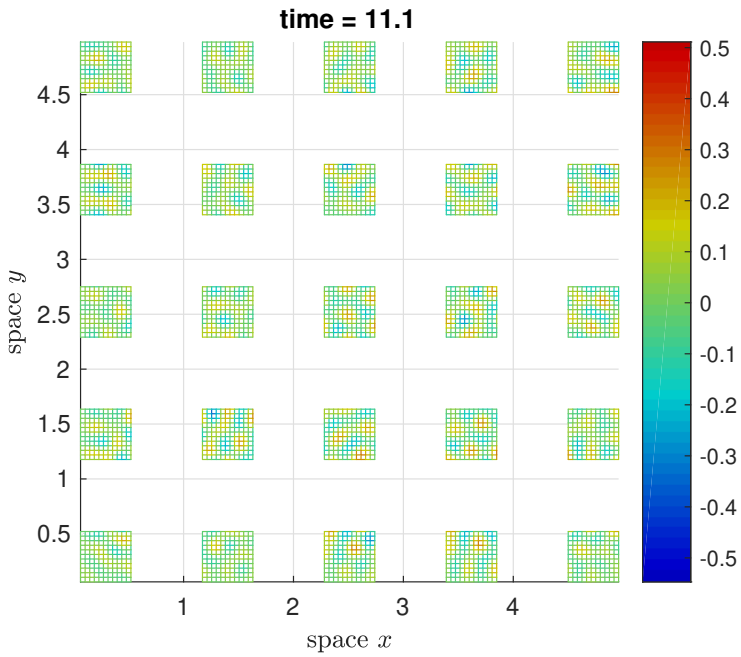


Figure 4.10: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. At this early time much of the random sub-patch microstrucre has decayed leaving some random marginal modes starting to grow.

```
ordCC = 4;
dom.type='equispace';
dom.bcOffset=[0.5 0.5;1.0 1.5]
patches = configPatches2(@SwiftHohenbergPDE,[0 Len],dom ...
    ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
xs=squeeze(patches.x);
ys=squeeze(patches.y);
```

4.10.0.1 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems. Integrate the interface `patchSys2` (Section 4.2) to the microscale differential equations (despite the extreme stiffness, `ode23` is ten times quicker than `ode15s`). Because pattern evolution is eventually phase-diffusion, here sample the pattern at quadratically varying times.

```
fprintf('\n**** Simulate in time\n')
u0 = 0.3*( -1+2*rand(size(patches.x+patches.y)) );
Ts=400*linspace(0,1,97).^2;
tic
[ts,us] = ode23(@patchSys2, Ts, u0(:),[],patches,k0,Ra);
simulateTime = toc
us = reshape(us',nSubP,nSubP,nPatch,nPatch,[]);
```

Plot the simulation such as that shown in Figures 4.10 to 4.15 First, reshape the data, omitting edge values.

```
xs([1:2 end-1:end],:) = nan;
ys([1:2 end-1:end],:) = nan;
us = reshape( permute(us,[1 3 2 4 5]) ...
```

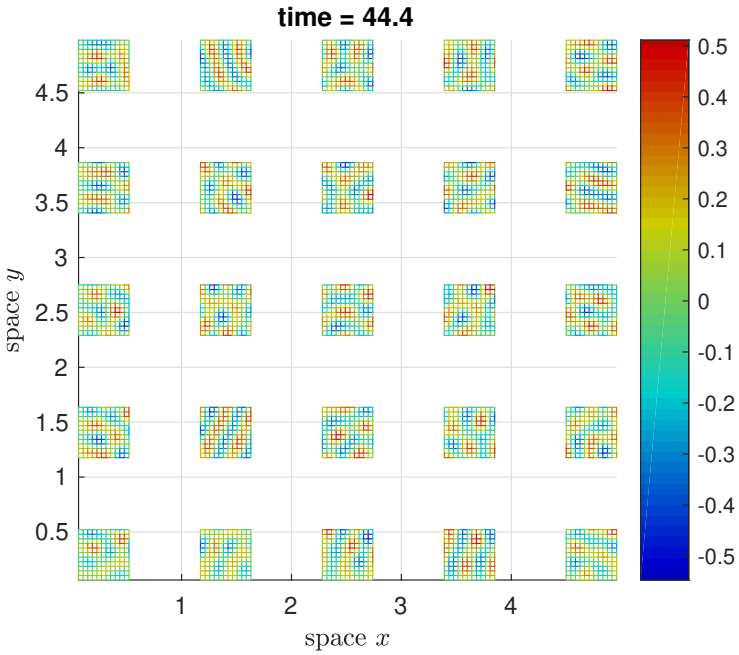


Figure 4.11: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. By now the local sub-patch patterns have reached a quasi-equilibrium amplitude.

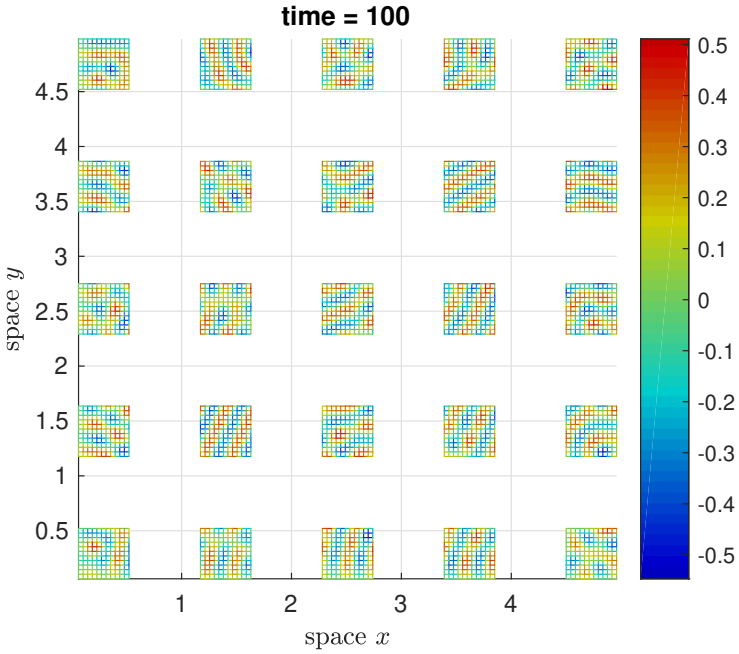


Figure 4.12: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Patterns within the patches are evolving to the preferred rolls, but with weak coupling to other patches.

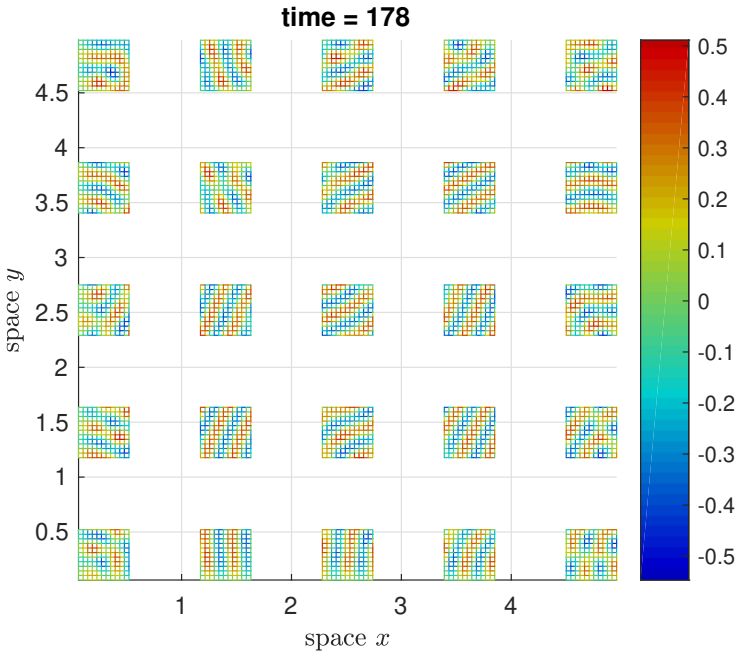


Figure 4.13: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Can see different effects arising at different types of boundaries.

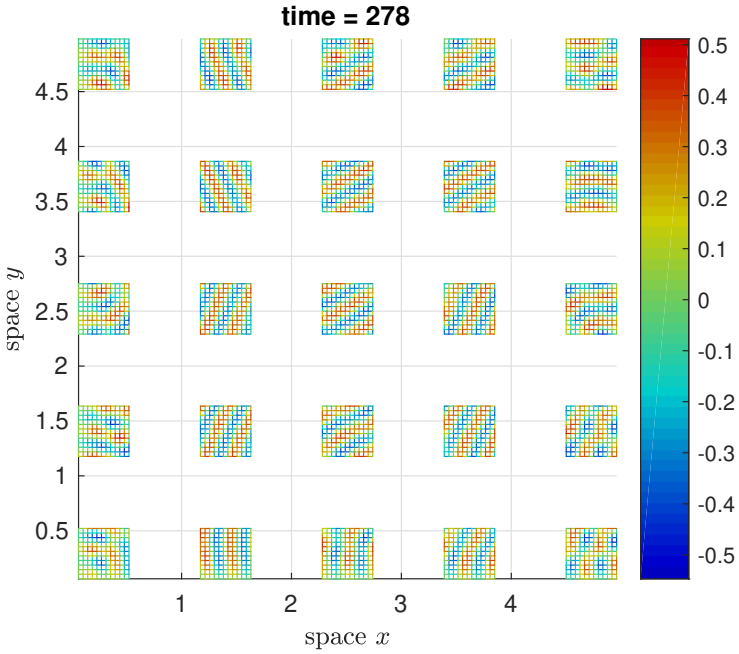


Figure 4.14: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. ...

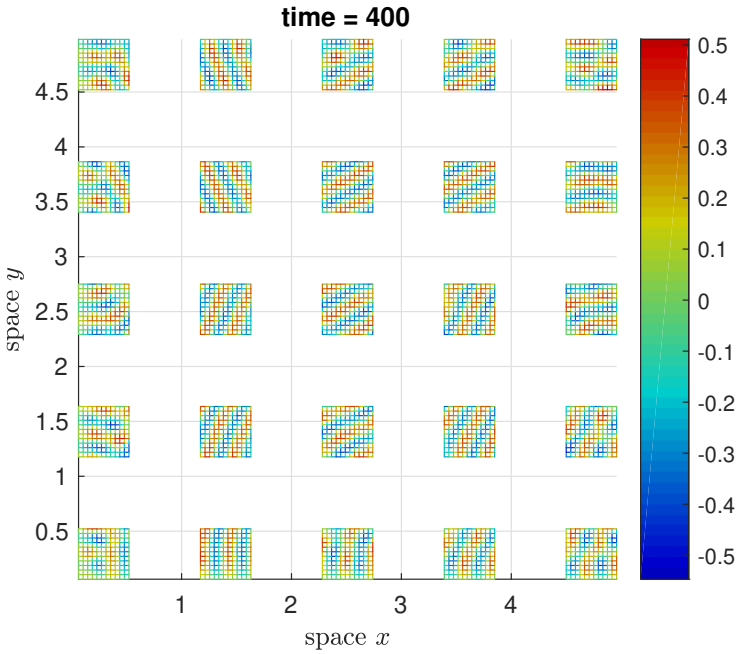


Figure 4.15: pattern field $u(x, y, t)$ in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. ...

```
138                ,nSubP*nPatch,nSubP*nPatch,[]);
139 uRange=[min(us(:)) max(us(:))];
```

Second, plot six examples of the evolving pattern, equi-spaced in time-index.

```
146 plots = round( 1+linspace(0,1,7)*(numel(ts)-1) )
147 for p=2:numel(plots)
148   figure(p),clf
149   mesh(xs(:),ys(:),us(:,:,plots(p))')
150   axis equal, view(0,90)
151   caxis(uRange), colormap(cMap), colorbar
152   xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y,t)$')
153   title(['time = ' num2str(ts(plots(p)),3)])
154   ifOurCf2eps([basename num2str(p)],[12 11])
155 end%for p
```

Third, plot animation in time: starts after a key press.

```
161 %%
162 figure(1),clf
163 cf=mesh(xs(:),ys(:),us(:,:,1)');
164 axis equal, view(0,90)
165 caxis(uRange), colormap(cMap), colorbar
166 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y,t)$')
167 title(['time = ' num2str(ts(1),3)])
168 ca=gca;
169 disp('Press any key to start animation'),pause
170 for p=2:numel(ts)
171   cf.ZData=us(:,:,p)';
172   cf.CData=us(:,:,p)';
173   ca.Title.String=['time = ' num2str(ts(p),3)];
```

```
174   pause(0.1)
175 end
```

Fin.

4.10.1 The Swift–Hohenberg PDE and BCs inside patches

As a microscale discretisation of Swift–Hohenberg PDE $u_t = -(1+\nabla^2/k_0^2)^2 u + \mathrm{Ra}\, u - u^3$, here code straightforward centred discretisation in space.

```
189 function ut=SwiftHohenbergPDE(t,u,patches,k0,Ra)
190   dx=diff(patches.x(1:2));  % microscale spacing
191   dy=diff(patches.y(1:2));  % microscale spacing
192   i=3:size(u,1)-2;  % interior points in patches
193   j=3:size(u,2)-2;  % interior points in patches
```

Code various boundary conditions. For slightly simpler coding, squeeze out the two singleton dimensions.

```
200   u = squeeze(u);
201   u(1:2,:,1,:)=0; % u=u_x=0 at x=0
202   u(:,1:2,:,1)=0; % u=u_y=0 at y=0
203   u(end-1,:,end,:)=0;                  % u=0   at x=L
204   u(end  ,:,end,:)=-u(end-2,:,end,:); % u_x=0 at x=L
205   u(:,end-1,:,end)=-u(:,end-2,:,end); % u_y=0   at y=L
206   u(:,end  ,:,end)=-u(:,end-3,:,end); % u_yyy=0 at y=L
```

Here code straightforward centred discretisation in space.

```
212   ut=nan+u;          % preallocate output array
213   v = u(2:end-1,2:end-1,:,:) ...
214       +( diff(u(:,2:end-1,:,:),2,1)/dx^2 ...
215         +diff(u(2:end-1,:,:,:),2,2)/dy^2 )/k0^2;
216   ut(i,j,:,:) = -( v(2:end-1,2:end-1,:,:) ...
217       +( diff(v(:,2:end-1,:,:),2,1)/dx^2 ...
218         +diff(v(2:end-1,:,:,:),2,2)/dy^2 )/k0^2 ) ...
219     +Ra*u(i,j,:,:) -u(i,j,:,:).^3;
220 end
```

5 Patches in 3D space

5.1 `configPatches3()`: configures spatial patches in 3D

Section contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys3()`, and possibly other patch functions. Sections 5.1.1 and 5.4 list examples of its use.

```
20 function patches = configPatches3(fun,Xlim,Dom ...
21     ,nPatch,ordCC,dx,nSubP,varargin)
22 version = '2023-04-12';
```

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 5.1.1 for an example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes time-derivatives (or time-steps) of quantities on the 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the rectangular-cuboid macro-space domain of the computation: namely $[$`Xlim(1)`$,$`Xlim(2)`$] \times [$`Xlim(3)`$,$`Xlim(4)`$ \times [$`Xlim(5)`$,$`Xlim(6)`$]$. If `Xlim` has two elements, then the domain is the cubic domain of the same interval in all three directions.
- `Dom` sets the type of macroscale conditions for the patches, and reflects the type of microscale boundary conditions of the problem. If `Dom` is `NaN` or `[]`, then the field `u` is triply macro-periodic in the 3D spatial domain, and resolved on equi-spaced patches. If `Dom` is a character string, then that specifies the `.type` of the following structure, with `.bcOffset` set to the default zero. Otherwise `Dom` is a structure with the following components.
  - `.type`, string, of either `'periodic'` (the default), `'equispace'`, `'chebyshev'`, `'usergiven'`. For all cases except `'periodic'`, users *must* code into `fun` the micro-grid boundary conditions that apply at the left/right/bottom/top/back/front faces of the left-most/rightmost/bottommost/topmost/backmost/frontmost patches, respectively.
  - `.bcOffset`, optional one, three or six element vector/array, in the cases of `'equispace'` or `'chebyshev'` the patches are placed so the left/right macroscale boundaries are aligned to the left/right faces of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when the micro-code applies Dirichlet boundary values on the extreme face micro-grid points, whereas use `bcOffset=0.5` when the microcode applies Neumann boundary conditions halfway

between the extreme face micro-grid points. Similarly for the top, bottom, back, and front faces.

If `.bcOffset` is a scalar, then apply the same offset to all boundaries. If three elements, then apply the first offset to both x-boundaries, the second offset to both y-boundaries, and the third offset to both z-boundaries. If six elements, then apply the first two offsets to the respective x-boundaries, the middle two offsets to the respective y-boundaries, and the last two offsets to the respective z-boundaries.

- `.X`, optional vector/array with `nPatch(1)` elements, in the case `'usergiven'` it specifies the x-locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Y`, optional vector/array with `nPatch(2)` elements, in the case `'usergiven'` it specifies the y-locations of the centres of the patches—the user is responsible the locations makes sense.
- `.Z`, optional vector/array with `nPatch(3)` elements, in the case `'usergiven'` it specifies the z-locations of the centres of the patches—the user is responsible the locations makes sense.

- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in all three directions, otherwise `nPatch(1:3)` gives the number (≥ 1) of patches in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the face-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.
- `dx` (real—scalar or three elements) is usually the sub-patch micro-grid spacing in x, y and z. If scalar, then use the same `dx` in all three directions, otherwise `dx(1:3)` gives the spacing in each of the three directions.

  However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio` (scalar or three elements), namely the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the equi-spacing of the patch mid-points—adjusted a little when `nEdge` > 1. So either `ratio` $= \frac{1}{2}$ means the patches abut and `ratio` $= 1$ is overlapping patches as in holistic discretisation, or `ratio` $= 1$ means the patches abut. Small `ratio` should greatly reduce computational time.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in all three directions, otherwise `nSubP(1:3)` gives the number in each direction. If not using `EdgyInt`, then `nSubP./nEdge` must be odd integer(s) so that there is/are centre-patch lattice planes. So for the defaults of `nEdge` $= 1$ and not `EdgyInt`, then `nSubP` must be odd.
- `'nEdge'`, *optional* (integer—scalar or three element), default=1, the width of face values set by interpolation at the face regions of each patch. If two elements, then respectively the width in x, y-directions.

The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `'EdgyInt'`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom/front/back face-values from right/left/bottom/top/back/front next-to-face values. If false or omitted, then interpolate from centre-patch planes.
- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `'hetCoeffs'`, *optional*, default empty. Supply a 3D or 4D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times m_z \times n_c$, where n_c is the number of different arrays of coefficients. For example, in heterogeneous diffusion, $n_c = 3$ for the diffusivities in the *three* different spatial directions (or $n_c = 6$ for the diffusivity tensor). The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the n_c coefficients being n_c parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1, 1)$-point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of `(nSubP-nEdge)/2` for not `EdgyInt`.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y \cdot m_z$ and construct an ensemble of all $m_x \cdot m_y \cdot m_z$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in x, y, z-directions, respectively, then this coupling cunningly preserves symmetry.
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

  If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension x, y, z corresponding to the highest `nPatch` (if a tie, then chooses the rightmost of x, y, z). A user may correspondingly distribute arrays with property `patches.codist`, or simply use formulas invoking the preset distributed arrays `patches.x`, `patches.y`, and `patches.z`. If a user has not yet established a parallel pool, then a 'local' pool is started.

Output The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct

available as a global variable.[1]

```
225 if nargout==0, global patches, end
226 patches.version = version;
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC` $\times$ 3-array of weights for the inter-patch interpolation onto the right/top/front and left/bottom/back faces (respectively) with patch:macroscale ratio as specified or as derived from `dx`.
- `.x` (8D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1 \times 1$ array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (8D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the regular spatial locations z_{kK} of the microscale grid points in every patch.
- `.ratio` 1×3, only for macro-periodic conditions, are the size ratios of every patch.
- `.nEdge` 1×3, is the width of face values set by interpolation at the face regions of each patch, in the x, y, z-directions respectively.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)`-1)$\times$(`nSubP(2)`-1)$\times$(`nSubP(3)`-1)$\times n_c$ 4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1$) $\times$ (`nSubP(2)` $- 1$) $\times$ (`nSubP(3)` $- 1$) $\times n_c \times m_x m_y m_z$ 5D array of $m_x m_y m_z$ ensemble of phase-shifts of the microscale heterogeneous coefficients.

[1] When using `spmd` parallel computing, it is generally best to avoid global variables, and so instead prefer using an explicit output variable.

- `.parallel`, logical: true if patches are distributed over multiple CPUs/ cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

5.1.1 If no arguments, then execute an example

```
315 if nargin==0
316 disp('With no arguments, simulate example of heterogeneous wave')
```

The code here shows one way to get started: a user's script may have the following three steps (" $\mapsto$ " denotes function recursion).

1. configPatches3
2. ode23 integrator $\mapsto$ patchSys3 $\mapsto$ user's PDE
3. process results

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the macro-wave time scale is roughly one.

```
334 mPeriod = [2 2 2];
335 cHetr = exp(0.9*randn([mPeriod 3]));
336 cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on $[-\pi, \pi]^3$-periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with 4^3 points forming each patch.

```
348 global patches
349 patches = configPatches3(@heteroWave3,[-pi pi] ...
350     ,'periodic' , 5, 0, 0.22, mPeriod+2 ,'EdgyInt',true ...
351     ,'hetCoeffs',cHetr);
```

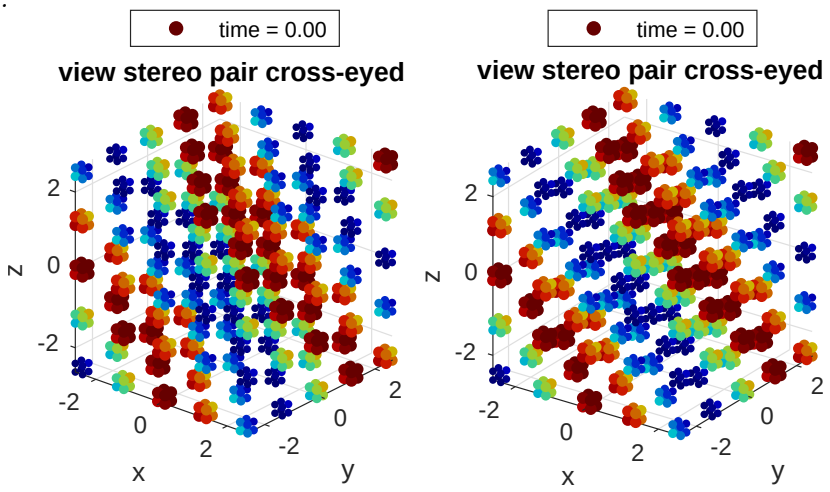
Set a wave initial state using auto-replication of the spatial grid, and as Figure 5.1 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

```
361 u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
362 v0 =    -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
363 uv0 = cat(4,u0,v0);
```

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean et al. 2020, Fig. 4).

```
380 disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
381 if ~exist('OCTAVE_VERSION','builtin')
382     [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
383 else %disp('octave version is very slow for me')
```

Figure 5.1: initial field $u(x,y,z,t)$ at time $t=0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 5.2 plots the computed field at time $t=6$.*




```
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 5.2. Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

Replicate x, y, and z arrays to get individual spatial coordinates of every data point. Then, optionally, set faces to `nan` so the plot just shows patch-interior data.

```
figure(1), clf, colormap(0.8*jet)
xs = patches.x+0*patches.y+0*patches.z;
ys = patches.y+0*patches.x+0*patches.z;
zs = patches.z+0*patches.y+0*patches.x;
if 1, xs([1 end],:,:,:)=nan;
      xs(:,[1 end],:,:)=nan;
      xs(:,:,[1 end],:)=nan;
end;%option
j=find(~isnan(xs));
```

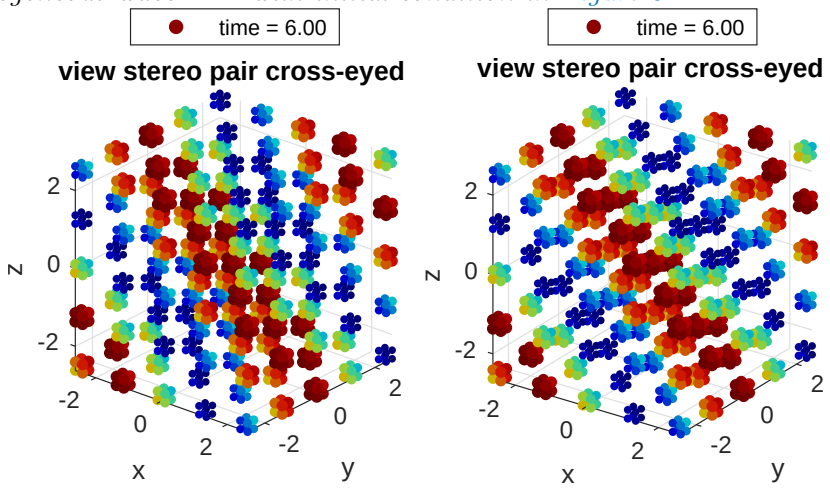
In the scatter plot, these functions `pix()` and `col()` map the u-data values to the size of the dots and to the colour of the dots, respectively.

```
pix = @(u) 15*abs(u)+7;
col = @(u) sign(u).*abs(u);
```

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
```

Figure 5.2: field $u(x, y, z, t)$ at time $t = 6$ of the patch scheme applied to the heterogeneous wave PDE *with initial condition in Figure 5.1.*




```
427   for p=1:2
428     subplot(1,2,p)
429     if (i==1)| exist('OCTAVE_VERSION','builtin')
430       scat(p) = scatter3(xs(j),ys(j),zs(j),'filled');
431       axis equal, caxis(col([0 1])), view(45-5*p,25)
432       xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
433       title('view stereo pair cross-eyed')
434     end % in matlab just update values
435     set(scat(p),'CData',col(u(j)) ...
436        ,'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
437     legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
438   end
```

Optionally save the initial condition to graphic file for Figure 4.1, and optionally save the last plot.

```
446   if i==1,
447     ifOurCf2eps([mfilename 'ic'])
448     disp('Type space character to animate simulation')
449     pause
450   else pause(0.05)
451   end
452 end% i-loop over all times
453 ifOurCf2eps([mfilename 'fin'])
```

Upon finishing execution of the example, exit this function.

```
468 return
469 end%if no arguments
```

5.1.2 `heteroWave3()`: heterogeneous Waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE is

$$u_t = v, \quad v_t = \vec{\nabla}(C\vec{\nabla} \cdot u)$$

for diagonal matrix C which has microscale variations. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSys3`, Section 5.2), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of heterogeneous coefficients, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
26 function ut = heteroWave3(t,u,patches)
27   if nargin<3, global patches, end
```

Microscale space-steps, and interior point indices.

```
33   dx = diff(patches.x(2:3));  % x micro-scale step
34   dy = diff(patches.y(2:3));  % y micro-scale step
35   dz = diff(patches.z(2:3));  % z micro-scale step
36   i = 2:size(u,1)-1; % x interior points in a patch
37   j = 2:size(u,2)-1; % y interior points in a patch
38   k = 2:size(u,3)-1; % z interior points in a patch
```

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
46   ut = nan+u;  % preallocate output array
47   ut(i,j,k,1,:) = u(i,j,k,2,:);
48   ut(i,j,k,2,:) ...
49   =diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,1,:),1),1)/dx^2 ...
50   +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,1,:),1,2),1,2)/dy^2 ...
51   +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,1,:),1,3),1,3)/dz^2;
52 end% function
```

5.2 patchSys3(): interface 3D space to time integrators

To simulate in time with 3D spatial patches we often need to interface a users time derivative function with time integration routines such as ode23 or PIRK2. This function provides an interface. Communicate patch-design variables (Section 5.1) either via the global struct patches or via an optional third argument. patches is required for the parallel computing of spmd, or if parameters are to be passed though to the user microscale function.

```
23 function dudt = patchSys3(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

Input

- u is a vector/array of length prod(nSubP)·nVars·nEnsem·prod(nPatch) where there are nVars · nEnsem field values at each of the points in the nSubP(1)×nSubP(2)×nSubP(3)×nPatch(1)×nPatch(2)×nPatch(3) spatial grid.
- t is the current time to be passed to the user's time derivative function.
- patches a struct set by configPatches3() with the following information used here.
  - .fun is the name of the user's function fun(t,u,patches,...) that computes the time derivatives on the patchy lattice. The array u has size nSubP(1)×nSubP(2)×nSubP(3)×nVars×nEsem× nPatch(1) × nPatch(2) × nPatch(3). Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
  - .x is nSubP(1) × 1 × 1 × 1 × 1nPatch(1) × 1 × 1 array of the spatial locations x_i of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - .y is similarly 1 × nSubP(2) × 1 × 1 × 1 × 1 × nPatch(2) × 1 array of the spatial locations y_j of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - .z is similarly 1 × 1 × nSubP(3) × 1 × 1 × 1 × 1 × nPatch(3) array of the spatial locations z_k of the microscale (i, j, k)-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
- varargin, optional, is arbitrary list of parameters to be passed onto the users time-derivative function as specified in configPatches3.

Output

- `dudt` is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length `prod(nSubP)` · `nVars` · `nEnsem` · `prod(nPatch)` and the same dimensions as `u`.

5.3 patchEdgeInt3(): sets 3D patch face values from 3D macroscale interpolation

Couples 3D patches across 3D space by computing their face values via macroscale interpolation. Assumes patch face values are determined by macroscale interpolation of the patch centre-plane values (Roberts et al. 2014, Bunder et al. 2021), or patch next-to-face values which appears better (Bunder et al. 2020). This function is primarily used by `patchSys3()` but is also useful for user graphics. [2]

Communicate patch-design variables via a second argument (optional, except required for parallel computing of `spmd`), or otherwise via the global struct `patches`.

```
27 function u = patchEdgeInt3(u,patches)
28 if nargin<2, global patches, end
```

Input

- `u` is a vector/array of length `prod(nSubP)·nVars·nEnsem·prod(nPatch)` where there are `nVars · nEnsem` field values at each of the points in the `nSubP1·nSubP2·nSubP3·nPatch1·nPatch2·nPatch3` multiscale spatial grid on the `nPatch1 · nPatch2 · nPatch3` array of patches.
- `patches` a struct set by `configPatches3()` which includes the following information.
  - `.x` is `nSubP1` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch1` $\times 1 \times 1$ array of the spatial locations x_{iI} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index i, but may be variable spaced in macroscale index I.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch2` $\times 1$ array of the spatial locations y_{jJ} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index j, but may be variable spaced in macroscale index J.
  - `.z` is similarly $1 \times 1 \times$ `nSubP3` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch3` array of the spatial locations z_{kK} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on the microscale index k, but may be variable spaced in macroscale index K.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.periodic` indicates whether macroscale is periodic domain, or alternatively that the macroscale has left, right, top, bottom, front and back boundaries so interpolation is via divided differences.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.

[2] Script `patchEdgeInt3test.m` verifies this code.

- `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in each of the x, y, z-directions—when invoking a periodic domain.
- `.EdgyInt`, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often preserves symmetry); false, from centre cross-patch values (near original scheme).
- `.nEdge`, three elements, the width of edge values set by interpolation at the x, y, z-face regions, respectively, of each patch (default is one all x, y, z-faces).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

- `u` is 8D array, `nSubP1` · `nSubP2` · `nSubP3` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2` · `nPatch3`, of the fields with face values set by interpolation.

5.4 homoDiffEdgy3: computational homogenisation of a 3D diffusion via simulation on small patches

Simulate heterogeneous diffusion in 3D space on 3D patches as an example application. Then compute macroscale eigenvalues of the patch scheme applied to this heterogeneous diffusion to validate and to compare various orders of inter-patch interpolation.

This code extends to 3D the 2D code discussed in **??**. First set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
29 mPeriod = randi([2 3],1,3)
30 cHetr = exp(0.3*randn([mPeriod 3]));
31 cHetr = cHetr*mean(1./cHetr(:))
```

Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Use spectral interpolation as we test other orders subsequently. In 3D we appear to get only real eigenvalues by using edgy interpolation. What happens for non-edgy interpolation is unknown.

```
42 nSubP=mPeriod+2;
43 nPatch=[5 5 5];
44 configPatches3(@heteroDiff3, [-pi pi], nan, nPatch ...
45     ,0, 0.3, nSubP, 'EdgyInt',true  ...
46     ,'hetCoeffs',cHetr );
```

5.4.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 5.3.

```
56 global patches
57 u0 = exp(-patches.x.^2/4-patches.y.^2/2-patches.z.^2);
58 u0 = u0.*(1+0.3*rand(size(u0)));
```

Integrate using standard integrators, unevenly spaced in time to better display transients.

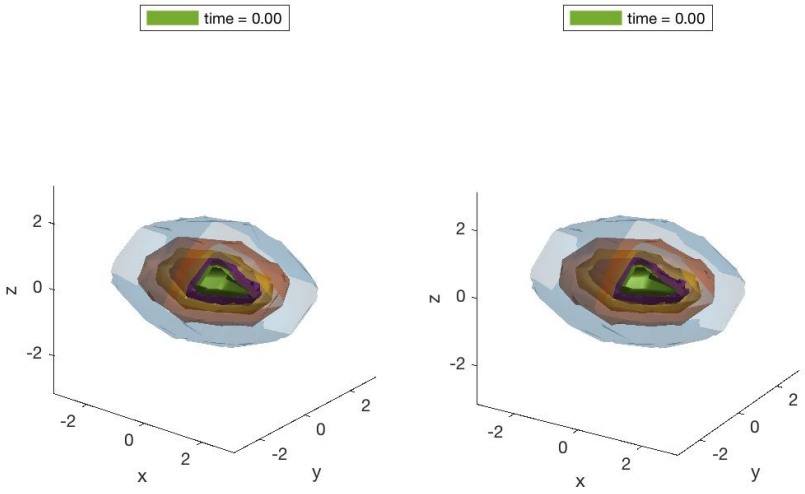
```
76 if ~exist('OCTAVE_VERSION','builtin')
77     [ts,us] = ode23(@patchSys3, 0.3*linspace(0,1,50).^2, u0(:));
78 else % octave version
79     [ts,us] = odeOcts(@patchSys3, 0.3*linspace(0,1).^2, u0(:));
80 end
```

Plot the solution as an animation over time.

```
88 figure(1), clf
89 rgb=get(gca,'defaultAxesColorOrder');
90 colormap(0.8*hsv)
```

Get spatial coordinates of patch interiors.

Figure 5.3: initial field $u(x, y, z, 0)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside. Figure 5.4 plots the isosurfaces of the computed field at time $t = 0.3$.*




```
96  x = reshape( patches.x([2:end-1],:,:,:) ,[],1);
97  y = reshape( patches.y(:,[2:end-1],:,:) ,[],1);
98  z = reshape( patches.z(:,:,[2:end-1],:) ,[],1);
```

For every time step draw the surface and pause for a short display.

```
105 for i = 1:length(ts)
```

Get the row vector of data, form into a 6D array, then omit patch faces, and reshape to suit the isosurface function. We do not use interpolation to get face values as the interpolation omits the corner edges and so breaks up the isosurfaces.

```
115   u = reshape( us(i,:) ,[nSubP nPatch]);
116   u = u([2:end-1],[2:end-1],[2:end-1],:,:,:);
117   u = reshape( permute(u,[1 4 2 5 3 6]) ...
118        , [numel(x) numel(y) numel(z)]);
```

Optionally cut-out the front corner so we can see inside.

```
124   u( (x>0) & (y'<0) & (shiftdim(z,-2)>0) ) = nan;
```

The `isosurface` function requires us to transpose x and y.

```
131 v = permute(u,[2 1 3]);
```

Draw cross-eyed stereo view of some isosurfaces.

```
137     clf;
138     for p=1:2
139       subplot(1,2,p)
140       for iso=5:-1:1
141          isov=(iso-0.5)/5;
142          hsurf(iso) = patch(isosurface(x,y,z,v,isov));
143          isonormals(x,y,z,v,hsurf(iso))
144          set(hsurf(iso) ,'FaceColor',rgb(iso,:) ...
145              ,'EdgeColor','none' ...
146              ,'FaceAlpha',iso/5);
147          hold on
148       end
149       axis equal, view(45-7*p,25)
150       axis(pi*[-1 1 -1 1 -1 1])
151       xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
152       legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
153       camlight, lighting gouraud
154       hold off
155     end% each p
156     if i==1 % pause for the viewer
157          makeJpeg=false;
158          if makeJpeg, print(['Figs/' mfilename 't0'],'-djpeg'), end
159          disp('Press any key to start animation of isosurfaces')
160          pause
161     else pause(0.05)
162     end
```

Finish the animation loop, and optionally output the isosurfaces of the final field, Figure 5.4.

```
178 end%for over time
179 if makeJpeg, print(['Figs/' mfilename 'tFin'],'-djpeg'), end
```

5.4.2 Compute Jacobian and its spectrum

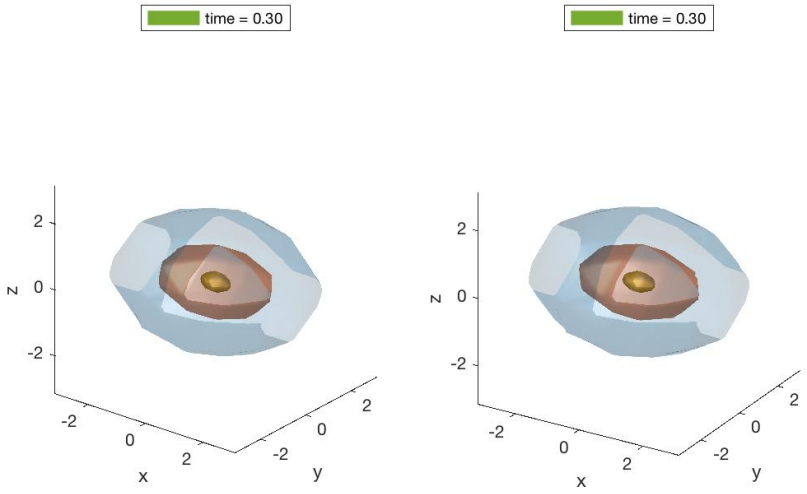
Let's explore the Jacobian dynamics for a range of orders of interpolation, all for the same random patch design and heterogeneity. Except here use a small ratio as we do not plot and then the scale separation is clearest.

```
195 ratio = 0.025*(1+rand(1,3))
196 nSubP=randi([3 5],1,3)
197 nPatch=[3 3 3]
198 nEnsem = prod(mPeriod) % or just set one
```

Find which elements of the 8D array are interior micro-grid points and hence correspond to dynamical variables.

```
205     u0 = zeros([nSubP,1,nEnsem,nPatch]);
206     u0([1 end],:,:,:) = nan;
207     u0(:,[1 end],:,:) = nan;
208     u0(:,:,[1 end],:) = nan;
```

Figure 5.4: final field $u(x, y, z, 0.3)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside.*




```
209     i = find(~isnan(u0));
210     sizeJacobian = length(i)
211     assert(sizeJacobian<4000 ...
212         ,'Jacobian is too big to quickly generate and analyse')
```

Store this many eigenvalues in array across different orders of interpolation.

```
219 nLeadEvals=prod(nPatch)+max(nPatch);
220 leadingEvals=[];
```

Evaluate eigenvalues for spectral as the base case for polynomial interpolation of order $2, 4, \ldots$.

```
228 maxords=6;
229 for ord=0:2:maxords
230     ord=ord
```

Configure with same heterogeneity.

```
236     configPatches3(@heteroDiff3,[-pi pi],nan,nPatch ...
237         ,ord,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
238         ,'hetCoeffs',cHetr);
```

Construct the Jacobian of the scheme as the matrix of the linear transformation, obtained by transforming the standard unit vectors.

```
246     jac = nan(length(i));
247     for j = 1:length(i)
```

```
248         u = u0(:)+(i(j)==(1:numel(u0))');
249         tmp = patchSys3(0,u);
250         jac(:,j) = tmp(i);
251       end
```

Test for symmetry, with error if we know it should be symmetric.

```
258       notSymmetric=norm(jac-jac')
259   %    if notSymmetric>1e-7, spy(abs(jac-jac')>1e-7), end%??
260       assert(notSymmetric<1e-7,'failed symmetry')
```

Find all the eigenvalues (as `eigs` is unreliable), and put eigenvalues in a vector.

```
267       [evecs,evals] = eig((jac+jac')/2,'vector');
268       biggestImag=max(abs(imag(evals)));
269       if biggestImag>0, biggestImag=biggestImag, end
```

Sort eigenvalues on their real-part with most positive first, and most negative last. Store the leading eigenvalues in `egs`, and write out when computed all orders. The number of zero eigenvalues, `nZeroEv`, gives the number of decoupled systems in this patch configuration.

```
279       [~,k] = sort(-real(evals));
280       evals=evals(k); evecs=evecs(:,k);
281       if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
282       if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
283       else % find evec closest to that of each leading spectral
284           [~,k]=max(abs(evecs'*evec0));
285           evals=evals(k); % re-sort in corresponding order
286       end
287       leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
288   end
289   disp('     spectral    quadratic      quartic  sixth-order ...')
290   leadingEvals=leadingEvals
```

5.4.3 `heteroDiff3()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSys3`, Section 5.2), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4+D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
23  function ut = heteroDiff3(t,u,patches)
24    if nargin<3, global patches, end
```

Microscale space-steps. Q: is using `i,j,k` slower than `2:end-1`??

```
31    dx = diff(patches.x(2:3));  % x micro-scale step
32    dy = diff(patches.y(2:3));  % y micro-scale step
```

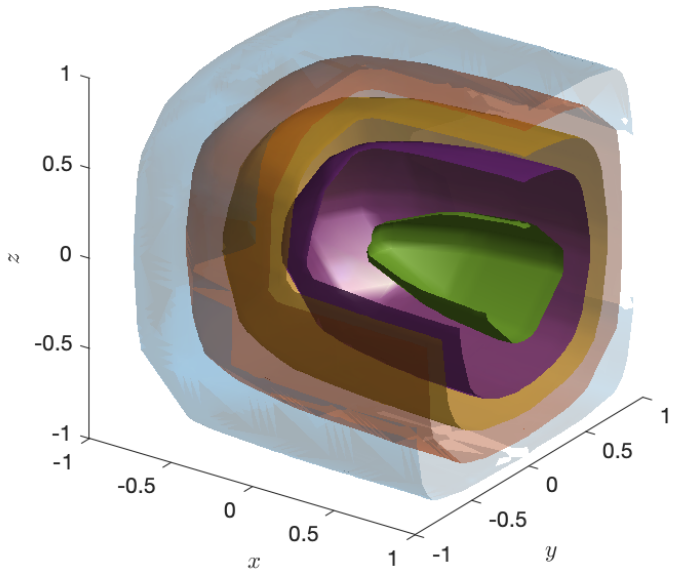
```
33    dz = diff(patches.z(2:3));  % z micro-scale step
34    i = 2:size(u,1)-1; % x interior points in a patch
35    j = 2:size(u,2)-1; % y interior points in a patch
36    k = 2:size(u,3)-1; % y interior points in a patch
```

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
44    ut = nan+u; % reserve storage
45    ut(i,j,k,:,:,:,:,:) ...
46    = diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,:,:,:,:,:),1),1)/dx^2 ...
47     +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,:,:,:,:,:),1,2),1,2)/dy^2 ...
48     +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,:,:,:,:,:),1,3),1,3)/dz^2;
49  end% function
```

Fin.

Figure 5.5: macroscale of the random heterogeneous diffusion in 3D with boundary conditions of zero on all faces except for the Neumann condition on $x = 1$ (Section 5.5). The small patches are equispaced in space.



5.5 `homoDiffBdryEquil3`: equilibrium via computational homogenisation of a 3D heterogeneous diffusion on small patches

Find the equilibrium of a forced heterogeneous diffusion in 3D space on 3D patches as an example application. Boundary conditions are Neumann on the right face of the cube, and Dirichlet on the other faces. Figure 5.5 shows five isosurfaces of the 3D solution field.

Clear variables, and establish globals.

```
33 clear all
34 global patches
35 %global OurCf2eps, OurCf2eps=true %option to save plots
```

Set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
46 mPeriod = randi([2 3],1,3)
47 cDiff = exp(0.3*randn([mPeriod 3]));
48 cDiff = cDiff*mean(1./cDiff(:))
```

Configure the patch scheme with some arbitrary choices of cubic domain, patches, and micro-grid spacing 0.05. Use high order interpolation as few patches in each direction. Configure for Dirichlet boundaries except for Neumann on the right x-face.

```
nSubP = mPeriod+2;
nPatch = 5;
Dom.type = 'equispace';
Dom.bcOffset = zeros(2,3);  Dom.bcOffset(2) = 0.5;
configPatches3(@microDiffBdry3, [-1 1], Dom ...
    , nPatch, 0, 0.05, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cDiff );
```

Set forcing, and store in global patches for access by the microcode

```
patches.fu = 10*exp(-patches.x.^2-patches.y.^2-patches.z.^2);
patches.fu = patches.fu.*(1+rand(size(patches.fu)));
```

Solve for steady state Set initial guess of zero, with `NaN` to indicate patch-edge values. Index `i` are the indices of patch-interior points, store in global patches for access by `theRes`, and the number of unknowns is then its number of elements.

```
u0 = zeros([nSubP,1,1,nPatch,nPatch,nPatch]);
u0([1 end],:,:,:) = nan;
u0(:,[1 end],:,:) = nan;
u0(:,:,[1 end],:) = nan;
patches.i = find(~isnan(u0));
nVariables = numel(patches.i)
```

Solve by iteration. Use `fsolve` for simplicity and robustness (optionally `optimoptions` to omit trace information), via the generic patch system wrapper `theRes` (Section 3.15).

```
disp('Solving system, takes 10--40 secs'),tic
uSoln = fsolve(@theRes,u0(patches.i) ...
        ,optimoptions('fsolve','Display','off'));
solveTime = toc
normResidual = norm(theRes(uSoln))
normSoln = norm(uSoln)
```

Store the solution into the patches, and give magnitudes.

```
u0(patches.i) = uSoln;
u0 = patchEdgeInt3(u0);
```

Plot isosurfaces of the solution

```
figure(1), clf
rgb=get(gca,'defaultAxesColorOrder');
```

Reshape spatial coordinates of patches.

```
x = patches.x(:);  y = patches.y(:);  z = patches.z(:);
```

Draw isosurfaces. Get the solution with interpolated faces, form into a 6D array, and reshape and transpose x and y to suit the isosurface function.

```
u = reshape( permute(squeeze(u0),[2 5 1 4 3 6]) ...
            , [numel(y) numel(x) numel(z)]);
maxu=max(u(:)),  minu=min(u(:))
```

Optionally cut-out the front corner so we can see inside.

```
  u( (x'>0) & (y<0) & (shiftdim(z,-2)>0) ) = nan;
```

Draw some isosurfaces.

```
clf;
for iso=5:-1:1
   isov=(iso-0.5)/5*(maxu-minu)+minu;
   hsurf(iso) = patch(isosurface(x,y,z,u,isov));
   isonormals(x,y,z,u,hsurf(iso))
   set(hsurf(iso) ,'FaceColor',rgb(iso,:) ...
       ,'EdgeColor','none' ,'FaceAlpha',iso/5);
   hold on
end
hold off
axis equal, axis([-1 1 -1 1 -1 1]), view(35,25)
xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
camlight, lighting gouraud
ifOurCf2eps(mfilename) %optionally save plot
if exist('OurCf2eps') && OurCf2eps,  print('-dpng',['Figs/' mfilename]), end
```

5.5.1 `microDiffBdry3()`: 3D forced heterogeneous diffusion with boundaries

This function codes the lattice forced heterogeneous diffusion inside the 3D patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSys3`, Section 5.2), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
function ut = microDiffBdry3(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps.

```
  dx = diff(patches.x(2:3));  % x micro-scale step
  dy = diff(patches.y(2:3));  % y micro-scale step
  dz = diff(patches.z(2:3));  % z micro-scale step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  k = 2:size(u,3)-1; % y interior points in a patch
```

Code microscale boundary conditions of say Neumann on right, and Dirichlet on left, top, bottom, front, and back (viewed along the z-axis).

```
  u( 1 ,:,:,:,:, 1 ,:,:)=0; %left face of leftmost patch
  u(end,:,:,:,:,end,:,:)=u(end-1,:,:,:,:,end,:,:); %right face of rightmost
```

```
213   u(:, 1 ,:,:,:,:, 1 ,:)=0; %bottom face of bottommost
214   u(:,end,:,:,:,:,end,:)=0; %top face of topmost
215   u(:,:, 1 ,:,:,:,:, 1 )=0; %front face of frontmost
216   u(:,:,end,:,:,:,:,end)=0; %back face of backmost
```

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
224   ut = nan+u; % reserve storage
225   ut(i,j,k,:) ...
226   = diff(patches.cs(:,j,k,1).*diff(u(:,j,k,:),1,1),1,1)/dx^2 ...
227    +diff(patches.cs(i,:,k,2).*diff(u(i,:,k,:),1,2),1,2)/dy^2 ...
228    +diff(patches.cs(i,j,:,3).*diff(u(i,j,:,:),1,3),1,3)/dz^2 ...
229    +patches.fu(i,j,k);
230 end% function
```

5.6 heteroDispersiveWave3: heterogeneous Dispersive Waves from 4th order PDE

This uses small spatial patches to simulate heterogeneous dispersive waves in 3D. The wave equation for $u(x, y, z, t)$ is the fourth-order in space PDE

$$u_{tt} = -\nabla^2(C\nabla^2 u)$$

for microscale variations in scalar $C(x, y, z)$.

Initialise some Matlab aspects.

```
clear all
cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
basename = [num2str(floor(1e5*rem(now,1))) mfilename]
%global OurCf2eps, OurCf2eps=true %optional to save plots
```

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the macro-wave time scale is roughly one.

```
mPeriod = [2 2 2];
cHetr = exp(0.9*randn(mPeriod));
cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a fourth-order heterogeneous wave PDE: to be solved on $[-\pi, \pi]^3$-periodic domain, with 5^3 patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with 6^3 points forming each patch. (Six because two edge layers on each of two faces, and two interior points for the PDE.)

```
global patches
patches = configPatches3(@heteroDispWave3,[-pi pi] ...
    ,'periodic', 5, 0, 0.22, mPeriod+4 ,'EdgyInt',true ...
    ,'hetCoeffs',cHetr ,'nEdge',2);
```

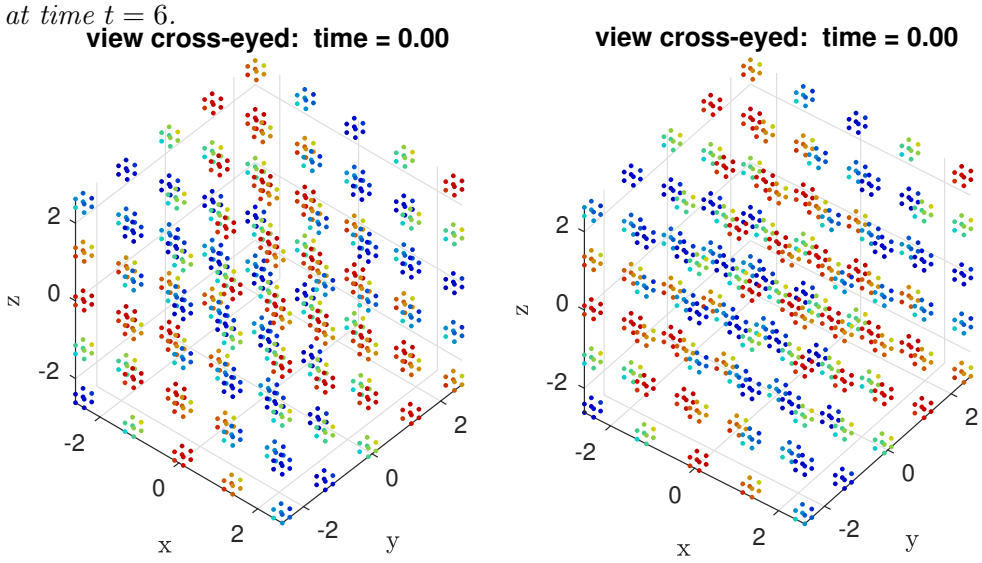
Set a wave initial state using auto-replication of the spatial grid, and as Figure 5.6 shows. This wave propagates diagonally across space. Concatenate the two u, v-fields to be the two components of the fourth dimension.

```
u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
v0 =    -0.5*cos(patches.x+patches.y+patches.z)*3;
uv0 = cat(4,u0,v0);
```

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean et al. 2020, Fig. 4).

```
disp('Simulate heterogeneous wave u_tt=delsq[C*delsq(u)]')
tic
[ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
simulateTime=toc
```

Figure 5.6: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous dispersive wave PDE*: Figure 5.7 plots the computed field at time $t = 6$.*



Animate the computed simulation to end with Figure 5.7. Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

Replicate x, y, and z arrays to get individual spatial coordinates of every data point. Then, optionally, set faces to `nan` so the plot just shows patch-interior data.

```
%%
figure(1), clf, colormap(cMap)
xs = patches.x+0*patches.y+0*patches.z;
ys = patches.y+0*patches.x+0*patches.z;
zs = patches.z+0*patches.y+0*patches.x;
if 1, xs([1:2 end-1:end],:,:,:)=nan;
      xs(:,[1:2 end-1:end],:,:)=nan;
      xs(:,:,[1:2 end-1:end],:)=nan;
end;%option
j=find(~isnan(xs));
```

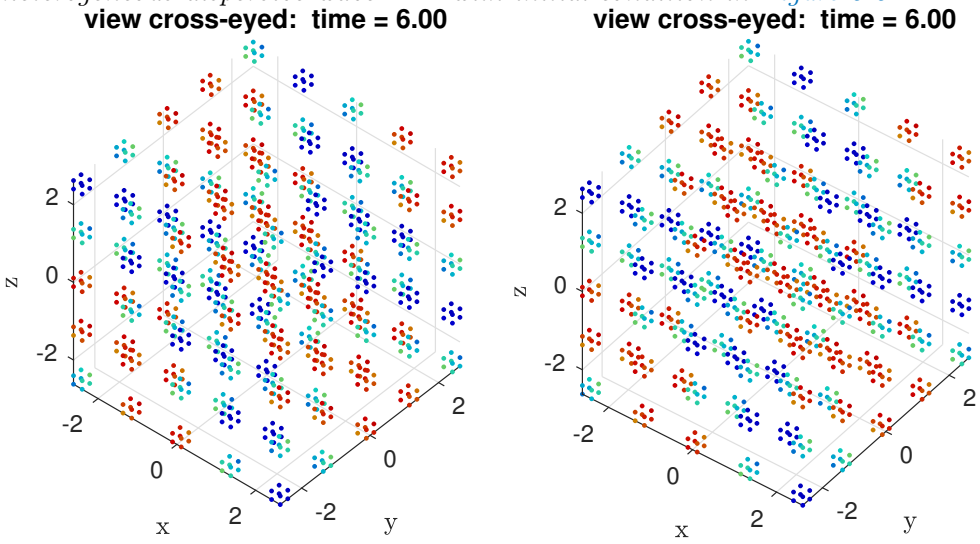
In the scatter plot, `col()` maps the u-data values to the colour of the dots.

```
col = @(u) sign(u).*abs(u);
```

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
  for p=1:2
    subplot(1,2,p)
    if (i==1)
      scat(p) = scatter3(xs(j),ys(j),zs(j),'.');
```

Figure 5.7: field $u(x, y, z, t)$ at time $t = 6$ of the patch scheme applied to the heterogeneous dispersive wave PDE *with initial condition in Figure 5.6.*




```
129         axis equal, caxis(col([0 1])), view(45-4*p,42)
130         xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
131       end
132       title(['view cross-eyed:  time = ' num2str(ts(i),'%4.2f')])
133       set( scat(p),'CData',col(u(j)) );
134     end
```

Optionally save the initial condition to graphic file for Figure 5.6, and optionally save the last plot.

```
142     if i==1,
143       ifOurCf2eps([basename 'ic'])
144       disp('Type space character to animate simulation')
145       pause
146     else pause(0.1)
147     end
148   end% i-loop over all times
149   ifOurCf2eps([basename 'fin'])
```

5.6.1 `heteroDispWave3()`: PDE function of 4th-order heterogeneous dispersive waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE for $u(x, y, z, t)$ and 'velocity' $v(x, y, z, t)$ is

$$u_t = v, \quad v_t = -\nabla^2(C\nabla^2 u)$$

for microscale variations in scalar $C(x, y, z)$. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSys3`, Section 5.2), computes the time derivative at each point in the interior of a patch, output in `ut`. The 3D array of heterogeneous coefficients, C_{ijk}, c^y_{ijk} and c^z_{ijk}, have been stored in `patches.cs` (3D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```
187 function ut = heteroDispWave3(t,u,patches)
188   if nargin<3, global patches, end
```

Micro-grid space steps.

```
194 dx = diff(patches.x(2:3));
195 dy = diff(patches.y(2:3));
196 dz = diff(patches.z(2:3));
```

First, compute $C\nabla^2 u$ into say `u`, using indices for all but extreme micro-grid points. We use a single colon to represent the last four array dimensions because the result arrays are already dimensioned.

```
205 I = 2:size(u,1)-1; J = 2:size(u,2)-1;  K = 2:size(u,3)-1;
206 u(I,J,K,1,:) = patches.cs(I,J,K,1,:).*( diff(u(:,J,K,1,:),2,1)/dx^2 ...
207     +diff(u(I,:,K,1,:),2,2)/dy^2 +diff(u(I,J,:,1,:),2,3)/dz^2 );
```

Reserve storage, set lowercase indices to non-edge interior, and then assign interior patch values to the heterogeneous diffusion time derivatives.

```
215 ut = nan+u;  % preallocate output array
216 i = I(2:end-1); j = J(2:end-1);  k = K(2:end-1);
217 ut(i,j,k,1,:) = u(i,j,k,2,:);  % du/dt=v
218 % dv/dt=delta^2 of above C*delta^2
219 ut(i,j,k,2,:) = -( diff(u(I,j,k,1,:),2,1)/dx^2 ...
220   +diff(u(i,J,k,1,:),2,2)/dy^2 +diff(u(i,j,K,1,:),2,3)/dz^2 );
221 end% function
```

6 Matlab parallel computation of the patch scheme

Chapter contents

For large-scale simulations, we here assume you have a compute cluster with many independent computer processors linked by a high-speed network. The functions we provide in our toolbox aim to distribute computations in parallel across the cluster. MATLAB*'s Parallel Computing Toolbox* empowers a reasonably straightforward way to implement this parallelisation.[1] The reason is that the patch scheme (Chapter 3) has a clear domain decomposition of assigning relatively few patches to each processor.

[1] This parallelisation is not written for, nor tested for, Octave.

The examples listed herein are all *Proof of Principle*: as coded they are all small enough that non-parallel execution is here much quicker that the parallel execution. One needs significantly larger and/or more detailed problems than these examples before parallel execution is effective.

As in all parallel cluster computing, interprocessor communication time all to often dominates. It is important to reduce communication as much as possible compared to computation. Consequently, parallel computing is only effective when there is a very large amount of microscale computation done on each processor per communication—all of the examples listed herein are quite small and so the parallel computation of these is much slower than serial computation. We guesstimate that the microscale code may need, per time-step, of the order of many millions of operations per processor in order for the parallelisation to be useful.

To help minimise communication in time-dependent problems we have drafted a special integrator `RK2mesoPatch`, Section 6.4, that communicates between patches only on a meso-time (Bunder et al. 2016).

6.1 `chanDispSpmd`: simulation of a 1D shear dispersion via simulation on small patches across a channel

Section contents

Simulate 1D shear dispersion along long thin channel, dispersion that is emergent from micro-scale dynamics in 2D space. Use 1D patches as a Proof of Principle example of parallel computing with `spmd`. In this shear dispersion, although the micro-scale diffusivities are one-ish, the shear causes an effective longitudinal 'diffusivity' of the order of Pe^2 —which is typically much larger than the micro-scale diffusivity (Taylor 1953, e.g.).

The spatial domain is the channel (large) L-periodic in x and $|y| < 1$. Seek to predict a concentration field $c(x, y, t)$ satisfying the linear advection-diffusion PDE

$$\frac{\partial c}{\partial t} = -\operatorname{Pe} u(y)\frac{\partial c}{\partial x} + \frac{\partial}{\partial x}\Big[\kappa_x(y)\frac{\partial c}{\partial x}\Big] + \frac{\partial}{\partial y}\Big[\kappa_y(y)\frac{\partial c}{\partial y}\Big]. \tag{6.1}$$

where Pe denotes a Peclet number, parabolic advection velocity $u(y) = \frac{3}{2}(1-y^2)$ with noise, and parabolic diffusivity $\kappa_x(y) = \kappa_y(y) = (1-y^2)$ with noise. The noise is to be multiplicative and log-normal to ensure advection and diffusion are all positive, and to be periodic in x.

For a microscale computation we discretise in space with x-spacing δx, and n_y points over $|y| < 1$ with spacing $\delta y := 2/n_y$ at $y_j := -1 + (j - \frac{1}{2})\delta y$, $j = 1 : n_y$. Our microscale discretisation of PDE (6.1) is then

$$\frac{\partial c_{ij}}{\partial t} = -\operatorname{Pe} u(y_j)\frac{c_{i+1,j} - c_{i-1,j}}{2\delta x} + \frac{d_{i,j+1/2} - d_{i,j-1/2}}{\delta y} + \frac{D_{i+1/2,j} - D_{i-1/2,j}}{\delta x},$$
$$d_{ij} := \kappa_y(y_j)\frac{c_{i,j+1/2} - c_{i,j-1/2}}{\delta y}, \quad D_{ij} := \kappa_x(y_j)\frac{c_{i+1/2,j} - c_{i-1/2,j}}{\delta x}. \tag{6.2}$$

These are coded in Section 6.1.4 for the computation.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive interprocessor communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 1
```

The micro-scale PDE is evaluated at positions y_j across the channel, $|y| < 1$. The even indexed points are the collocation points for the PDE, whereas the odd indexed points are the half-grid points for specification of y-diffusivities.

```
ny = 7
y = linspace(-1,1,2*ny+1);
yj = y(2:2:end);
```

Set micro-scale advection (array 1) and diffusivity (array 2) with (roughly) parabolic shape (Watt & Roberts 1995, MacKenzie & Roberts 2003, e.g.). Here modify the parabola by a heterogeneous log-normal factor with specified period along the channel: modify the strength of the heterogeneity by the coefficient of `randn` from zero to perhaps one: coefficient 0.3 appears a good moderate value. Remember that `configPatches1` reshapes `cHetr` to 2D.

```
mPeriod = 4
cHetr = shiftdim([3/2 1],-1).*(1-y.^2) ...
        .*exp(0.3*randn([mPeriod 2*ny+1 2]));
```

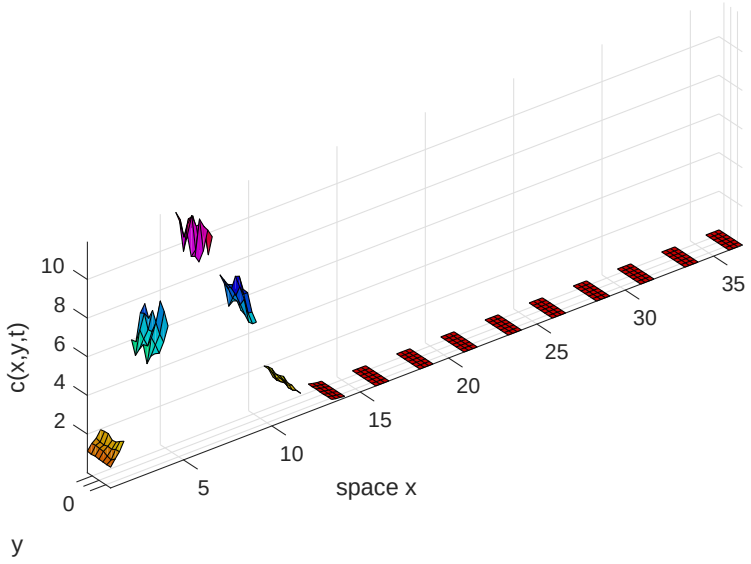
Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Choose some random order of interpolation to see the alternatives. Set `patches` information to be global so the info can be used for Cases 1–2 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
if theCase<=2, global patches, end
nPatch=15
nSubP=2+mPeriod
ratio=0.2+0.2*(theCase<4)
Len=nPatch/ratio
ordCC=2*randi([0 3])
disp('**** Setting configPatches1')
patches = configPatches1(@chanDispMicro, [0 Len], nan ...
    , nPatch, ordCC, ratio, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

When using parallel then additional parameters to `patches` should be set within a `spmd` block (because `patches` is a co-distributed structure).

```
Peclet = 10
if theCase==1, patches.Pe = Peclet;
else     spmd, patches.Pe = Peclet; end
end
```

Figure 6.1: initial field $u(x, y, 0)$ of the patch scheme applied to a heterogeneous advection-diffusion PDE. *Figure 6.2 plots the roughly smooth field values at time $t = 4$. In this example the patches are relatively large, ratio 0.4, for visibility.*



6.1.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 6.1.

```
149 disp('**** Set initial condition and test dc0dt =')
150 if theCase==1
```

Without parallel processing, invoke the usual operations.

```
156     c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
157     c0 = c0.*(1+0.2*rand(size(c0)));
158     dc0dt = patchSys1(0,c0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes use `patches.codist` to explicitly code how to distribute new arrays over the cpus. Now `patchSys1` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we *must* pass it explicitly as a parameter to `patchSys1`.

```
171 else, spmd
172     c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
173     c0 = c0.*(1+0.2*rand(size(c0),patches.codist));
174     dc0dt = patchSys1(0,c0,patches)
175     end%spmd
176 end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up

to 0.3, need micro-steps less than 0.005; with noise 1, need micro-steps less than 0.0015.

```
198 warning('Integrating system in time, wait patiently')
199 ts=4*linspace(0,1);
```

Go to the selected case.

```
205 switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys1` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
217 case 1
218 %    [cs,uerrs] = RK2mesoPatch(ts,c0);
219     [ts,cs] = ode23(@patchSys1,ts,c0(:));
220     cs=reshape(cs,[length(ts) size(c0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
232 case 2
233     cs = RK2mesoPatch(ts,c0);
234     cs = cs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
243 case 3,spmd
244     cs = RK2mesoPatch(ts,c0,[],patches);
245     end%spmd
246     cs = cs{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 6.3.3) to compute each burst of the micro-scale simulation. For a Peclet number of ten, the macro-scale time-step needs to be less than about 0.5 (which here is very little projection)—presumably the mean advection in a macro-step needs to be less than about the patch spacing. The function `microBurst()` here interfaces to `aBurst()` (Section 6.1.3) in order to provide shaped initial states, and to provide the patch information.

```
264 case 4
265     microBurst = @(tb0,xb0,bT) ...
266         aBurst(tb0 ,reshape(xb0,size(c0)) ,patches);
267     ts = 0:0.7:5
268     cs = PIRK2(microBurst,ts,gather(c0(:)));
269     cs = reshape(cs,[length(ts) size(c0)]);
```

End the four cases.

```
276 end%switch theCase
```

6.1.2 Plot the solution

Optionally set to save some plots to file.

```
287 if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the computed solution field over time

```
293 figure(1), clf, colormap(0.8*hsv)
```

First get the x-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
301 if theCase==1, x = patches.x;
302 else, spmd
303     x = gather( patches.x );
304     end%spmd
305     x = x{1};
306 end
307 x([1 end],:,:,:) = nan;
```

For every time step draw the concentration values as a set of surfaces on 2D patches, with a short pause to display animation.

```
315 nTimes = length(ts)
316 for l = 1:nTimes
```

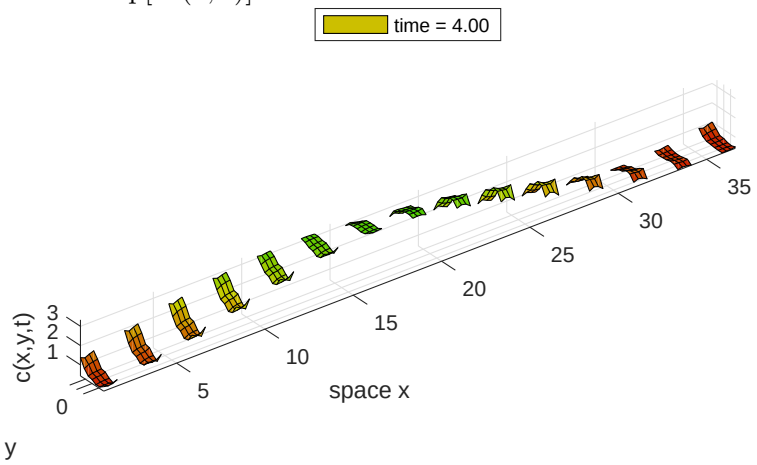
At each time, squeeze sub-patch data into a 3D array, permute to get all the x-variation in the first two dimensions, and reshape into x-variation for each and every (y).

```
325   c = reshape( permute( squeeze( ...
326       cs(l,:,:,:,:) ) ,[1 3 2]) ,numel(x),ny);
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space.

```
333   if l==1
334     hp = surf(x(:),yj,c');
335     axis([0 Len -1 1 0 max(c(:))])
336     axis equal
337     xlabel('space $x$'), ylabel('$y$'); zlabel('$c(x,y,t)$')
338     ifOurCf2eps([mfilename 't0'])
339     legend(['time = ' num2str(ts(l),'%4.2f')] ...
340         ,'Location','north')
341     disp('**** pausing, press blank to animate')
342     pause
343  else
344     hp.ZData = c';
345     legend(['time = ' num2str(ts(l),'%4.2f')])
```

Figure 6.2: final field $c(x, y, 4)$ of the patch scheme applied to a heterogeneous advection-diffusion PDE (6.1) *with heterogeneous factor log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
346      pause(0.1)
347    end
```

Finish the animation loop, and optionally save the final plot to file, Figure 6.2.

```
363 end%for over time
364 ifOurCf2eps([mfilename 'tFin'])
```

Macro-scale view Plot a macro-scale mesh of the predictions: at each of a selection of times, for every patch, plot the patch-mean value at the mean-x.

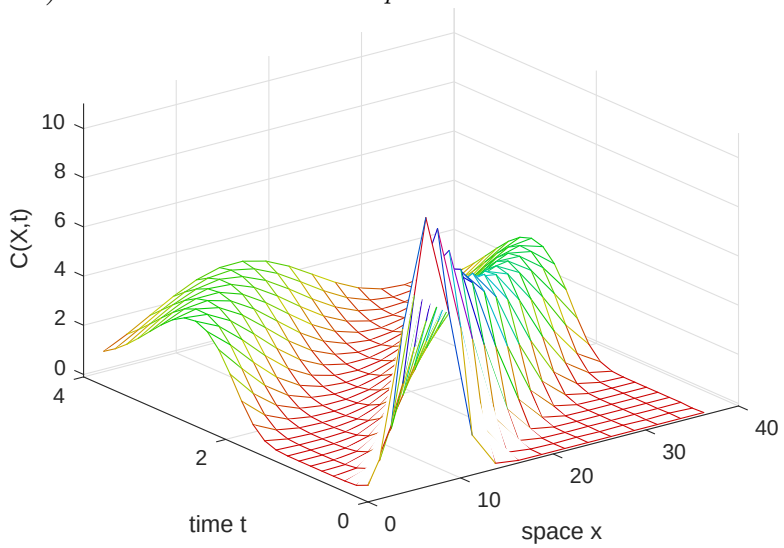
```
374 figure(2), clf, colormap(0.8*hsv)
375 X = squeeze(mean(x(2:end-1,:,:,:)));
376 C = squeeze(mean(mean(cs(:,2:end-1,:,:,:),2),3));
377 j = 1:ceil(nTimes/30):nTimes;
378 mesh(X,ts(j),C(j,:));
379 xlabel('space $x$'),ylabel('time $t$'),zlabel('$C(X,t)$')
380 zlim([-0.1 11])
381 ifOurCf2eps([mfilename 'Macro'])
```

6.1.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.2. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0033. With macro-step 0.5, these parameters usually give stable projective integration.

```
404 function [tbs,xbs] = aBurst(tb0,xb0,patches)
405     normx=max(abs(xb0(:)));
406     disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
407     assert(normx<20,'solution exploding')
408     tbs = tb0+(0:0.033:0.2);
```

Figure 6.3: macro-scale view of heterogeneous advection-diffusion PDE *along a (periodic) channel obtained via the patch scheme.*




```
409     spmd
410       xb0 = codistributed(xb0,patches.codist);
411       xbs = RK2mesoPatch(tbs,xb0,[],patches);
412     end%spmd
413     xbs=reshape(xbs{1},length(tbs),[]);
414 end%function
```

Fin.

6.1.4 `chanDispMicro()`: heterogeneous 2D advection-diffusion in a long thin channel

This function codes the lattice heterogeneous diffusion inside the patches. For 4D input arrays of concentration `c` and spatial lattice `x` (via edge-value interpolation of `patchSys1`, Section 3.2), computes the time derivative (6.2) at each point in the interior of a patch, output in `ct`. The heterogeneous advections and diffusivities, $u_i(y_j)$ and $\kappa_i(y_{j+1/2})$, have previously been merged and stored in the one array `patches.cs` (2D).

```
22 function ct = chanDispMicro(t,c,p)
23   [nx,ny,~,~]=size(c); % micro-grid points in patches
24   ix = 2:nx-1;         % x interior points in a patch
25   dx = diff(p.x(2:3)); % x space step
26   dy = 2/ny;           % y space step
27   ct = nan+c;          % preallocate output array
28   pcs = reshape(p.cs,nx-1,[],2);
```

Compute the cross-channel flux using 'ghost' nodes at channel boundaries, so that the flux is zero at $y = \pm 1$ either because the boundary values are replicated so the differences are zero, or because the diffusivities in `cs` are zero at the channel boundaries.

```
38    ydif = pcs(ix,1:2:end,2) ...
39           .*(c(ix,[1:end end],:,:)-c(ix,[1 1:end],:,:))/dy;
```

Now evaluate advection-diffusion time derivative (6.2). Could use upwind advection and no longitudinal diffusion, or, as here, centred advection and diffusion.

```
48    ct(ix,:,:,:) = (ydif(:,2:end,:,:)-ydif(:,1:end-1,:,:))/dy ...
49    + diff(pcs(:,2:2:end,2).*diff(c))/dx^2 ...
50    - p.Pe*pcs(ix,2:2:end,1).*(c(ix+1,:,:,:)-c(ix-1,:,:,:))/(2*dx);
51  end% function
```

6.2 `rotFilmSpmd`: simulation of a 2D shallow water flow on a rotating heterogeneous substrate

Section contents

As an example application, consider the flow of a shallow layer of fluid on a solid flat rotating substrate, such as in spin coating (Wilson et al. 2000, Oron et al. 1997, §II.K, e.g.) or large-scale shallow water waves (Dellar & Salmon 2005, Hereman 2009, e.g.). Let $\vec{x} = (x, y)$ parametrise location on the rotating substrate, and let the fluid layer have thickness $h(\vec{x}, t)$ and move with depth-averaged horizontal velocity $\vec{v}(\vec{x}, t) = (u, v)$. We take as given (with its simplified physics) that the (non-dimensional) governing set of PDEs is the nonlinear system (Bunder & Roberts 2018, eq. (1), e.g.)

$$\frac{\partial h}{\partial t} = -\nabla \cdot (h\vec{v}), \tag{6.3a}$$

$$\frac{\partial \vec{v}}{\partial t} = \begin{bmatrix} -b & f \\ -f & -b \end{bmatrix} \vec{v} - (\vec{v}\cdot\nabla)\vec{v} - g\nabla h + \vec{\nabla}\cdot(\nu\vec{\nabla}\vec{v})\,, \tag{6.3b}$$

where $b(\vec{x})$ represents heterogeneous 'bed' drag, f is the Coriolis coefficient, g is the acceleration due to gravity, $\nu(\vec{x})$ is a heterogeneous 'kinematic viscosity', and we neglect surface tension.

The aim is to simulate the macroscale dynamics which (for constant b) is approximately that of the nonlinear diffusion $\partial h/\partial t \approx \frac{gb}{b^2+f^2}\vec{\nabla}\cdot(h\vec{\nabla}h)$ (Bunder & Roberts 2018, eq. (2)). But there is no known algebraic closure for the macroscale in the case of heterogeneous $b(\vec{x})$ and $\nu(\vec{x})$, nonetheless the patch scheme automatically predicts a sensible macroscale for such heterogeneous dynamics (Figure 6.5).

For the microscale computation, Section 6.2.4 discretises the PDEs (6.3) in space with x, y-spacing $\delta x, \delta y$.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
71 clear all
72 theCase = 1
```

Set micro-scale bed drag (array 1) and diffusivity (arrays 2–3) to be a heterogeneous log-normal factor with specified period: modify the strength of the heterogeneity by the coefficient of `randn` from zero to perhaps one: coefficient 0.3 appears a good moderate value.

```
82 mPeriod = 5
83 bnu = shiftdim([1 0.5 0.5],-1) ...
84       .*exp(0.3*randn([mPeriod mPeriod 3]));
```

Configure the patch scheme with these choices of domain, patches, size ratios—here each patch is square in space. In Cases 1–2, set `patches` information to be global so the info can be used without being explicitly passed as arguments.

```
96 if theCase<=2, global patches, end
```

In Case 4, double the size of the domain and use more separated patches accordingly, to maintain the spatial microscale grid spacing to be 0.055. Here use fourth order edge-based coupling between patches. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
108 nSubP = 2+mPeriod
109 nPatch = 9
110 ratio = 0.2+0.2*(theCase<4)
111 Len = 2*pi*(1+(theCase==4))
112 disp('**** Setting configPatches2')
113 patches = configPatches2(@rotFilmMicro, [0 Len], nan ...
114     , nPatch, 4, ratio, nSubP, 'EdgyInt',true  ...
115     ,'hetCoeffs',bnu ,'parallel',(theCase>1) );
```

When using parallel, any additional parameters to `patches`, such as physical parameters for the microcode, must be set within a `spmd` block (because `patches` is a co-distributed structure). Here set frequency of substrate rotation, and strength of gravity.

```
125 f = 5,  g = 1
126 if theCase==1, patches.f = f; patches.g = g;
127 else     spmd, patches.f = f; patches.g = g; end
128 end
```

6.2.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 6.4. Here the initial condition is a (periodic) quasi-Gaussian in h and zero velocity $\vec{v}$, with additive

random perturbations.

```
141 disp('**** Set initial condition and test dhuv0dt =')
142 if theCase==1
```

When not parallel processing, invoke the usual operations. Here add a random noise to the velocity field, but keep $h(x, y, 0)$ smooth as shown by Figure 6.4. The `shiftdim(...,-1)` moves the given row-vector of coefficients into the third dimension to become coefficients of the fields (h, u, v), respectively.

```
153     huv0 = shiftdim([0.5 0 0],-1) ...
154         .*exp(-cos(patches.x)/2-cos(patches.y));
155     huv0 = huv0+0.1*shiftdim([0 1 1],-1).*rand(size(huv0));
156     dhuv0dt = patchSys2(0,huv0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in Cases 2–4 here. Also, we must sometimes explicitly tell functions how to distribute some initial condition arrays over the cpus. Now `patchSys2` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside an `spmd`-block we *must* pass `patches` explicitly as a parameter to `patchSys2`.

```
170 else, spmd
171     huv0 = shiftdim([0.5 0 0],-1) ...
172         .*exp(-cos(patches.x)/2-cos(patches.y));
173     huv0 = huv0+0.1*rand(size(huv0),patches.codist);
174     dhuv0dt = patchSys2(0,huv0,patches)
175     end%spmd
176 end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up to 0.3, need micro-steps less than 0.0003; with noise 1, need micro-steps less than 0.0001.

```
201 warning('Integrating system in time, wait a minute')
202 ts=0:0.003:0.3;
```

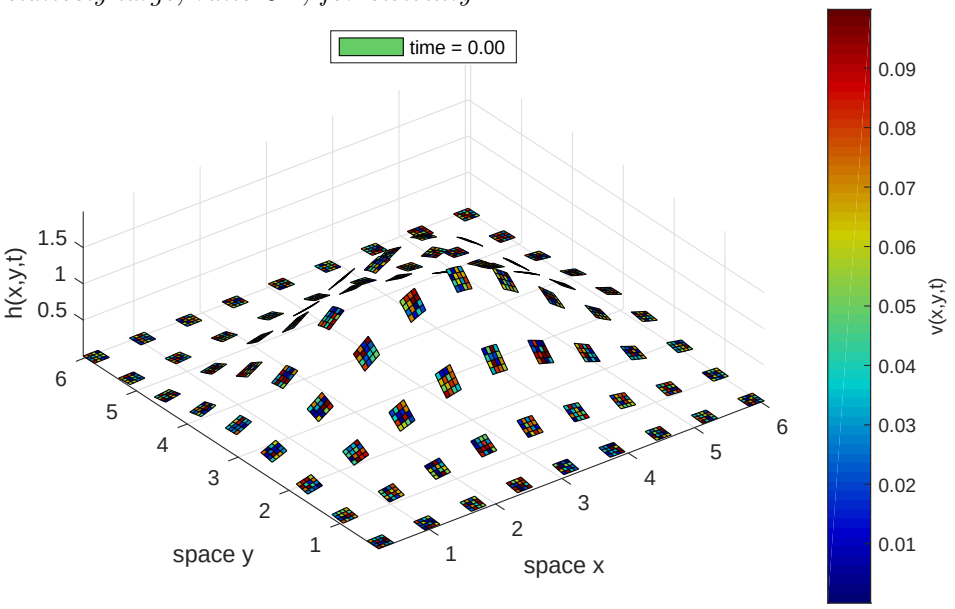
Go to the selected case.

```
208 switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys2` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
220 case 1
221 %    tic,[huvs,uerrs] = RK2mesoPatch(ts,huv0);toc
222      [ts,huvs] = ode23(@patchSys2,[0 4],huv0(:));
223      huvs=reshape(huvs,[length(ts) size(huv0)]);
```

Figure 6.4: initial field $h(x, y, 0)$ of the patch scheme applied to the heterogeneous, shallow water, rotating substrate, PDE (6.3). *The micro-scale sub-patch colour displays the initial y-direction velocity field $v(x, y, 0)$. Figure 6.5 plots the roughly smooth field values at time $t = 6$. In this example the patches are relatively large, ratio* 0.4*, for visibility.*



2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
235 case 2
236     huvs = RK2mesoPatch(ts,huv0);
237     huvs = huvs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
246 case 3,spmd
247     huvs = RK2mesoPatch(ts,huv0,[],patches);
248     end%spmd
249     huvs = huvs{1};
```

4. In this fourth case, use Projective Integration (PI). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 6.2.3) to compute each burst of the micro-scale simulation. The macro-scale time-step needs to be less than about 0.1 (which here is not much projection). The function `microBurst()` interfaces to `aBurst()` (Section 6.2.3) in order to provide shaped initial states, and to provide the patch information.

```
264 case 4
265     microBurst = @(tb0,xb0,bT) ...
```

```
266           aBurst(tb0 ,reshape(xb0,size(huv0)) ,patches);
267       ts = 0:0.1:1
268       huvs = PIRK2(microBurst,ts,gather(huv0(:)));
269       huvs = reshape(huvs,[length(ts) size(huv0)]);
```

End the four cases.

```
276 end%switch theCase
```

6.2.2 Plot the solution

Optionally set to save some plots to file.

```
287 if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the computed solution field over time

```
293 figure(1), clf, colormap(0.8*jet)
```

First get the x-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
300 if theCase==1, x = patches.x;
301                y = patches.y;
302 else, spmd
303     x = gather( patches.x );
304     y = gather( patches.y );
305     end%spmd
306     x = x{1}; y = y{1};
307 end
308 x([1 end],:,:,:,:,:) = nan;
309 y(:,[1 end],:,:,:,:) = nan;
```

Draw the field values as a patchy surface evolving over 100–200 time steps.

```
316 nTimes = length(ts)
317 for l = 1:ceil(nTimes/200):nTimes
```

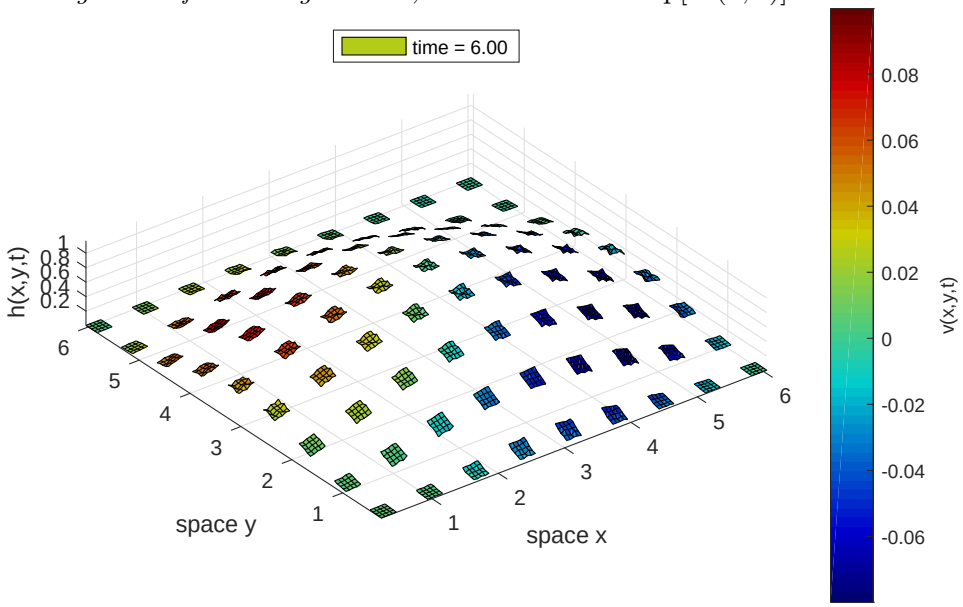
At each time, squeeze sub-patch data fields into three 4D arrays, permute to get all the x/y-variations in the first/last two dimensions, and and then reshape to 2D.

```
325   h = reshape( permute( squeeze( ...
326       huvs(l,:,:,1,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
327   u = reshape( permute( squeeze( ...
328       huvs(l,:,:,2,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
329   v = reshape( permute( squeeze( ...
330       huvs(l,:,:,3,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space. Colour the surface according to the velocity v in the y-direction.

```
338   if l==1
339     hp = surf(x(:),y(:),h',v');
```

Figure 6.5: final field $h(x, y, 6)$, coloured by $v(x, y, 6)$, of the patch scheme applied to the heterogeneous, shallow water, rotating substrate, PDE (6.3) *with heterogeneous factors log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
340      axis([0 Len 0 Len 0 max(h(:))])
341      c = colorbar; c.Label.String = 'v(x,y,t)';
342      legend(['time = ' num2str(ts(1),'%4.2f')] ...
343          ,'Location','north')
344      axis equal
345      xlabel('space $x$'), ylabel('space $y$'), zlabel('$h(x,y,t)$')
346      ifOurCf2eps([mfilename 't0'])
347      disp('**** pausing, press blank to begin animation')
348      pause
349    else
350      hp.ZData = h'; hp.CData = v';
351      legend(['time = ' num2str(ts(1),'%4.2f')])
352      pause(0.1)
353    end
```

Finish the animation loop, and optionally save the final plot to file, Figure 6.5.

```
370  end%for over time
371  ifOurCf2eps([mfilename 'tFin'])
```

6.2.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.01. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0003. With macro-step 0.1, these parameters usually give stable projective integration.

```
388 function [tbs,xbs] = aBurst(tb0,xb0,patches)
389     normx=max(abs(xb0(:)));
390     disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
391     assert(normx<20,'solution exploding')
392     tbs = tb0+(0:0.003:0.015);
393     spmd
394       xb0 = codistributed(xb0,patches.codist);
395       xbs = RK2mesoPatch(tbs,xb0,[],patches);
396     end%spmd
397     xbs=reshape(xbs{1},length(tbs),[]);
398 end%function
```

Fin.

6.2.4 `rotFilmMicro()`: 2D shallow water flow on a rotating heterogeneous substrate

This function codes the heterogeneous shallow water flow (6.3) inside 2D patches. The PDEs are discretised on the multiscale lattice in terms of evolving variables h_{ijIJ}, u_{ijIJ} and v_{ijIJ}. For 6D input array `huv` (via edge-value interpolation of `patchEdgeInt2()`, Section 4.2), computes the time derivatives (6.3) at each point in the interior of a patch, output in `huvt`. The heterogeneous bed drag and diffusivities, b_{ij} and ν_{ij}, have previously been merged and stored in the array `patches.cs` (2D $\times$ 3): herein `patches` is named `p`.

```
24 function huvt = rotFilmMicro(t,huv,p)
25   [nx,ny,~]=size(huv); % micro-grid points in patches
26   i = 2:nx-1;          % x interior points in a patch
27   j = 2:ny-1;          % y interior points in a patch
28   dx = diff(p.x(2:3)); % x space step
29   dy = diff(p.y(2:3)); % y space step
30   huvt = nan+huv;      % preallocate output array
```

Set indices of fields in the arrays. Need to store different diffusivity values for the x, y-directions as they are evaluated at different points in space.

```
38   h=1; u=2; v=3;
39   b=1; nux=2; nuy=3;
```

Use a staggered micro-grid so that `h(i,j)` $= h_{ij}$, `u(i,j)` $= u_{i+1/2,j}$, and `v(i,j)` $= v_{i,j+1/2}$. We need the following to interpolate some quantities to other points on the staggered micro-grid. But the first two statements fill-in two needed corner values because they are not (currently) interpolated by `patchEdgeInt2()`.

```
51 huv(1,ny,u,:,:,:) = huv(2,ny,u,:,:,:)+huv(1,ny-1,u,:,:,:) ...
52                     -huv(2,ny-1,u,:,:,:);
53 huv(nx,1,v,:,:,:) = huv(nx,2,v,:,:,:)+huv(nx-1,1,v,:,:,:) ...
54                     -huv(nx-1,2,v,:,:,:);
55 v4u = (huv(i,j-1,v,:,:,:)+huv(i+1,j,v,:,:,:) ...
56       +huv(i,j,v,:,:,:)+huv(i+1,j-1,v,:,:,:))/4;
```

```
u4v = (huv(i,j+1,u,:,:,:)+huv(i-1,j,u,:,:,:) ...
      +huv(i,j,u,:,:,:)+huv(i-1,j+1,u,:,:,:))/4;
h2u = (huv(2:nx,:,h,:,:,:)+huv(1:nx-1,:,h,:,:,:))/2;
h2v = (huv(:,2:ny,h,:,:,:)+huv(:,1:ny-1,h,:,:,:))/2;
```

Evaluate conservation of mass PDE (6.3a) (needing averages of h at half-grid points):

```
  huvt(i,j,h,:,:,:) = ...
    - (h2u(i,j  ,:,:,:,:).*huv(i  ,j,u,:,:,:) ...
      -h2u(i-1,j,:,:,:,:).*huv(i-1,j,u,:,:,:) )/dx ...
    - (h2v(i,j  ,:,:,:,:).*huv(i,j  ,v,:,:,:) ...
      -h2v(i,j-1,:,:,:,:).*huv(i,j-1,v,:,:,:))/dy ;
```

Evaluate the x-direction momentum PDE (6.3b) (needing to interpolate component v to u-points):

```
  huvt(i,j,u,:,:,:) = ...
    - p.cs(i,j,b).*huv(i,j,u,:,:,:) + p.f.*v4u ...
    - huv(i,j,u,:,:,:).*(huv(i+1,j,u,:,:,:)-huv(i-1,j,u,:,:,:))/(2*dx) ...
    - v4u.*(huv(i,j+1,u,:,:,:)-huv(i,j-1,u,:,:,:))/(2*dy) ...
    - p.g*(huv(i+1,j,h,:,:,:)-huv(i,j,h,:,:,:))/dx ...
    + diff(p.cs(:,j,nux).*diff(huv(:,j,u,:,:,:),[],1),[],1)/dx^2 ...
    + diff(p.cs(i,:,nuy).*diff(huv(i,:,u,:,:,:),[],2),[],2)/dy^2 ;
```

Evaluate the y-direction momentum PDE (6.3b) (needing to interpolate component u to v-points):

```
  huvt(i,j,v,:,:,:) = ...
    - p.cs(i,j,b).*huv(i,j,v,:,:,:) - p.f.*u4v ...
    - u4v.*(huv(i+1,j,v,:,:,:)-huv(i-1,j,v,:,:,:))/(2*dx) ...
    - huv(i,j,v,:,:,:).*(huv(i,j+1,v,:,:,:)-huv(i,j-1,v,:,:,:))/(2*dy) ...
    - p.g*(huv(i,j+1,h,:,:,:)-huv(i,j,h,:,:,:))/dy ...
    + diff(p.cs(:,j,nux).*diff(huv(:,j,v,:,:,:),[],1),[],1)/dx^2 ...
    + diff(p.cs(i,:,nuy).*diff(huv(i,:,v,:,:,:),[],2),[],2)/dy^2 ;
end% function
```

6.3 `homoDiff31spmd`: computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion

Section contents

Simulate effective dispersion along 1D space on 3D patches of heterogeneous diffusion as a Proof of Principle example of parallel computing with `spmd`. With only one patch in each of the y, z-directions, the solution simulated is strictly periodic in y, z with period `ratio`: there are only macro-scale variations in the x-direction. The discussion here only addresses issues with `spmd` parallel computing. For discussion on the 3D patch scheme with heterogeneous diffusion, see code and documentation for `homoDiffEdgy3` in Section 5.4.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` for minimising coding by a user of `spmd`-blocks;
- `theCase=3` is for users happier to explicitly invoke `spmd`-blocks.
- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
48 clear all
49 theCase = 1
```

Set micro-scale heterogeneity with various spatial periods in the three directions.

```
57 mPeriod = [4 3 2] %1+randperm(3)
58 cHetr = exp(0.3*randn([mPeriod 3]));
59 cHetr = cHetr*mean(1./cHetr(:))
```

Configure the patch scheme with some arbitrary choices of domain, patches, size ratios—here each patch is a unit cube in space. Choose some random order of interpolation. Set `patches` information to be global so the info can be used for Case 1 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
if any(theCase==[1 2]), global patches, end
nSubP=mPeriod+2
nPatch=[9 1 1]
ratio=0.3
Len=nPatch(1)/ratio
ordCC=2*randi([0 3])
disp('**** Setting configPatches3')
patches = configPatches3(@heteroDiff3,[0 Len 0 1 0 1], nan ...
    , nPatch, ordCC, [ratio 1 1], nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

6.3.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 6.6.

```
disp('**** Set initial condition and testing du0dt =')
if theCase==1
```

Without parallel processing, invoke the usual operations.

```
    u0 = exp( -(patches.x-Len/2).^2/Len ...
               -patches.y.^2/2-patches.z.^2 );
    u0 = u0.*(1+0.2*rand(size(u0)));
    du0dt = patchSys3(0,u0);
```

With parallel, must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes explicitly code how to distribute some new arrays over the cpus. Now `patchSys3` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we must pass it explicitly as a parameter to `patchSys3`.

```
else, spmd
    u0 = exp( -(patches.x-Len/2).^2/Len ...
               -patches.y.^2/2-patches.z.^2/4 );
    u0 = u0.*(1+0.2*rand(size(u0),patches.codist));
    du0dt = patchSys3(0,u0,patches);
    end%spmd
end%if theCase
```

Integrate in time. Use non-uniform time-steps for fun, and to show more of the initial rapid transients.

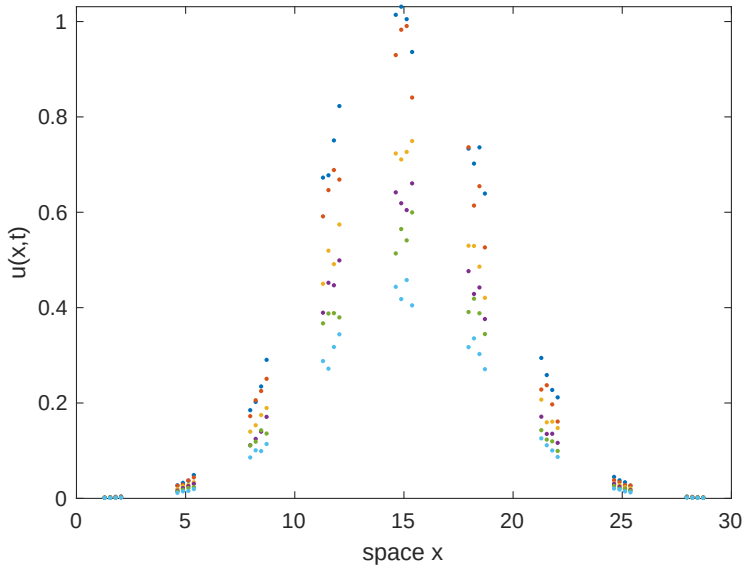
Alternatively, use `RK2mesoPatch` which reduces communication between patches, recalling that, by default, `RK2mesoPatch` does ten micro-steps for each specified step in `ts`. For unit cube patches, need micro-steps less than about 0.004 for stability.

```
warning('Integrating system in time, wait patiently')
ts=0.4*linspace(0,1,21).^2;
```

Go to the selected case.

```
switch theCase
```

Figure 6.6: initial field $u(x, y, z, 0)$ of the patch scheme applied to a heterogeneous diffusion PDE. *The vertical spread indicates the extent of the structure in u in the cross-section variables y, z. Figure 6.7 plots the nearly smooth field values at time $t = 0.4$.*



1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys3` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—that of the initial condition.

```
163 case 1
164 %    [us,uerrs] = RK2mesoPatch(ts,u0);
165      [ts,us] = ode23(@patchSys3,ts,u0(:));
166      us=reshape(us,[length(ts) size(u0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
178 case 2
179      us = RK2mesoPatch(ts,u0);
180      us = us{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
189 case 3,spmd
190      us = RK2mesoPatch(ts,u0,[],patches);
191      end%spmd
192      us = us{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel

spmd-blocks only invoked inside function aBurst() (Section 6.3.3) to compute each burst of the micro-scale simulation. A macro-scale time-step of about 3 seems good to resolve the decay of the macro-scale 'homogenised' diffusion. [2] The function microBurst() here interfaces to aBurst() (Section 6.3.3) in order to provide shaped initial states, and to provide the patch information.

```
case 4
    microBurst = @(tb0,xb0,bT) ...
        aBurst(tb0 ,reshape(xb0,size(u0)) ,patches);
    ts = 0:3:51
    us = PIRK2(microBurst,ts,gather(u0(:)));
    us = reshape(us,[length(ts) size(u0)]);
```

End the four cases.

```
end%switch theCase
```

6.3.2 Plot the solution

Optionally save some plots to file.

```
if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the solution field over time. Since the spatial domain is long in x and thin in y, z, just plot field values as a function of x.

```
figure(1), clf
if theCase==1
    x = reshape( patches.x(2:end-1,:,:,:) ,[],1);
else, spmd
    x = reshape(gather( patches.x(2:end-1,:,:,:) ),[],1);
    end%spmd
    x = x{1};
end
```

For every time step draw the field values as dots and pause for a short display.

```
nTimes = length(ts)
for l = 1:length(ts)
```

At each time, squeeze interior point data into a 4D array, permute to get all the x-variation in the first two dimensions, and reshape into x-variation for each and every (y, z).

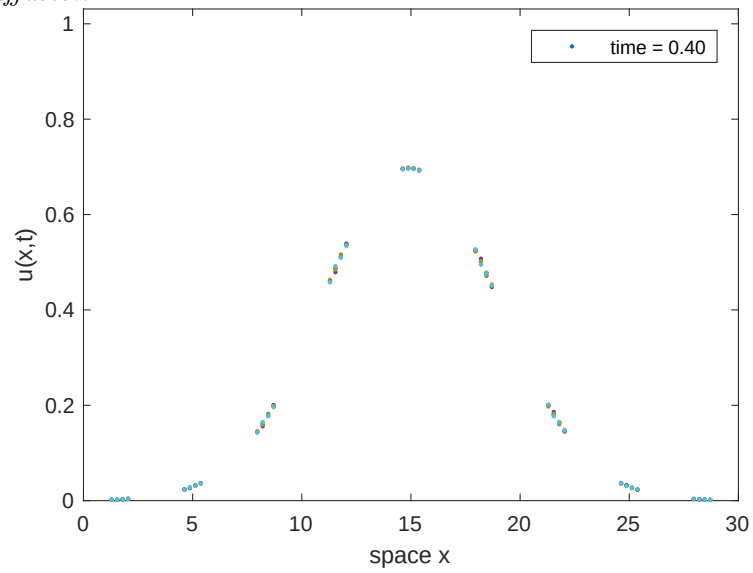
```
  u = reshape( permute( squeeze( ...
      us(l,2:end-1,2:end-1,2:end-1,:) ) ,[1 4 2 3]) ,numel(x),[]);
```

Draw point data to show spread at each cross-section, as well as macro-scale variation in the long space direction.

```
  if l==1
    hp = plot(x,u,'.');
```

[2] Curiously, PIG() appears to suffer unrecoverable instabilities with its variable step size!

Figure 6.7: final field $u(x, y, z, 0.4)$ of the patch scheme applied to a heterogeneous diffusion PDE.




```
275      axis([0 Len 0 max(u(:))])
276      xlabel('space $x$'), ylabel('$u(x,y,z,t)$')
277      ifOurCf2eps([mfilename 't0'])
278      legend(['time = ' num2str(ts(1),'%4.2f')])
279      disp('**** pausing, press blank to animate')
280      pause
281    else
282      for p=1:size(u,2), hp(p).YData=u(:,p); end
283      legend(['time = ' num2str(ts(1),'%4.2f')])
284      pause(0.1)
285    end
```

Finish the animation loop, and optionally output the final plot, Figure 6.7.

```
298  end%for over time
299  ifOurCf2eps([mfilename 'tFin'])
```

6.3.3 `microBurst` function for Projective Integration

Projective Integration stability seems to need bursts longer than 0.2. Here take ten meso-steps, each with default ten micro-steps so the micro-scale step is 0.002. With macro-step 3, these parameters usually give stable projective integration (but not always).

```
315  function [tbs,xbs] = aBurst(tb0,xb0,patches)
316      normx=max(abs(xb0(:)));
317      disp(['aBurst t = ' num2str(tb0) '  |x| = ' num2str(normx)])
318      assert(normx<10,'solution exploding')
319      tbs = tb0+(0:0.02:0.2);
320      spmd
321        xb0 = codistributed(xb0,patches.codist);
```

```
322         xbs = RK2mesoPatch(tbs,xb0,[],patches);
323       end%spmd
324       xbs=reshape(xbs{1},length(tbs),[]);
325   end%function
```

Fin.

6.4 RK2mesoPatch()

This is a Runge–Kutta, 2nd order, integration of a given deterministic system of ODEs on patches. It invokes meso-time updates of the patch-edge values in order to reduce interpolation costs, and uses a linear variation in edge-values over the meso-time-step (Bunder et al. 2016, case $Q = 2$). This function is aimed primarily for large problems executed on a computer cluster to markedly reduce expensive communication between computers.

If using within projective integration, it appears quite tricky to get all the time-steps chosen appropriately. One has to choose times for: the micro-scale time-step, the meso-time interval between communications, the longer meso-time burst length, and the macro-scale integration time-step.

```
27 function [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches)
28 if nargin<4, global patches, end
```

Input

- `patches.fun()` is a function such as `dxdt=fun(t,x,patches)` that computes the right-hand side of the ODE $d\vec{x}/dt = \vec{f}(t, \vec{x})$ where $\vec{x}$ is a vector/array, t is a scalar, and the result $\vec{f}$ is a correspondingly sized vector/array.
- `x0` is an vector/array of initial values at the time `ts(1)`.
- `ts` is a vector of meso-scale times to compute the approximate solution, say in $\mathbb{R}^\ell$ for $\ell \geq 2$.
- `nMicro`, optional, default 10, is the number of micro-time-steps taken for each meso-scale time-step.
- `patches` struct set by `configPatches`n and provided as either as parameter, or as a global variable.

Output

- `xs`, 5/7/9D (depending upon `nD`) array of length $\ell \times \cdots$ of approximate solution vector/array at the specified times. But, if using parallel computing via `spmd`, then `xs` is a *composite* 5/7/9D array, so outside of an `spmd`-block access a single copy of the array via `xs{1}`. Similarly for `errs`.
- `errs`, column vector in $\mathbb{R}^\ell$ of local error estimate for the step from t_{k-1} to t_k.

Bibliography

Abdulle, A., Arjmand, D. & Paganoni, E. (2020), A parabolic local problem with exponential decay of the resonance error for numerical homogenization, Technical report, Institute of Mathematics, École Polytechnique Fédérale de Lausanne.

Bunder, J., Divahar, J., Kevrekidis, I. G., Mattner, T. W. & Roberts, A. (2021), 'Large-scale simulation of shallow water waves with computation only on small staggered patches', *International Journal for Numerical Methods in Fluids* **93**(4), 953–977.

Bunder, J. E., Kevrekidis, I. G. & Roberts, A. J. (2020), Equation-free patch scheme for efficient computational homogenisation via self-adjoint coupling, Technical report, http://arxiv.org/abs/2007.06815.

Bunder, J. E. & Roberts, A. J. (2018), Nonlinear emergent macroscale PDEs, with error bound, for nonlinear microscale systems, Technical report, [https://arxiv.org/abs/1806.10297].

Bunder, J. E., Roberts, A. J. & Kevrekidis, I. G. (2017), 'Good coupling for the multiscale patch scheme on systems with microscale heterogeneity', *J. Computational Physics* **337**, 154–174.

Bunder, J., Roberts, A. J. & Kevrekidis, I. G. (2016), 'Accuracy of patch dynamics with mesoscale temporal coupling for efficient massively parallel simulations', *SIAM Journal on Scientific Computing* **38**(4), C335–C371.

Cao, M. & Roberts, A. J. (2016), 'Multiscale modelling couples patches of nonlinear wave-like simulations', *IMA J. Applied Maths.* **81**(2), 228–254.

Combescure, C. (2022), 'Selecting Generalized Continuum Theories for Nonlinear Periodic Solids Based on the Instabilities of the Underlying Microstructure', *Journal of Elasticity* .

Dellar, P. J. & Salmon, R. (2005), 'Shallow water equations with a complete coriolis force and topography', *Phys. Fluids* **17**, 106601.

Eckhardt, D. & Verfürth, B. (2022), Fully discrete heterogeneous multiscale method for parabolic problems with multiple spatial and temporal scales, Technical report, http://arxiv.org/abs/2210.04536.

Frewen, T. A., Hummer, G. & Kevrekidis, I. G. (2009), 'Exploration of effective potential landscapes using coarse reverse integration', *The Journal of Chemical Physics* **131**(13), 134104.

Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005*a*), 'Projecting to a slow manifold: singularly perturbed systems and legacy codes', *SIAM J. Applied Dynamical Systems* **4**(3), 711–732.
http://www.siam.org/journals/siads/4-3/60829.html

Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005*b*), 'Projecting to a slow manifold: Singularly perturbed systems and legacy codes', *SIAM Journal on Applied Dynamical Systems* **4**(3), 711–732.

Gear, C. W. & Kevrekidis, I. G. (2003*a*), 'Computing in the past with forward integration', *Phys. Lett. A* **321**, 335–343.

Gear, C. W. & Kevrekidis, I. G. (2003*b*), 'Projective methods for stiff differential equations: Problems with gaps in their eigenvalue spectrum', *SIAM Journal on Scientific Computing* **24**(4), 1091–1106.
http://link.aip.org/link/?SCE/24/1091/1

Gear, C. W. & Kevrekidis, I. G. (2003*c*), 'Telescopic projective methods for parabolic differential equations', *Journal of Computational Physics* **187**, 95–109.

Givon, D., Kevrekidis, I. G. & Kupferman, R. (2006), 'Strong convergence of projective integration schemes for singularly perturbed stochastic differential systems', *Comm. Math. Sci.* **4**(4), 707–729.

Govaerts, W., Kuznetsov, Y. A., Meijer, H., Al-Hdaibat, B., Witte, V. D., Dhooge, A., Mestrom, W., Neirynck, N., Riet, A. & Sautois, B. (2019), Matcont: Continuation toolbox for odes in matlab, Technical report, https://webspace.science.uu.nl/~kouzn101/NBA/ManualMatcontAug2019.pdf.

Hereman, W. (2009), Shallow water waves and solitary waves, *in* 'Mathematics of Complexity and Dynamical Systems', Springer, New York, pp. 8112–8125.

Hu, Z., Pan, G., Wang, Y. & Wu, Z. (2016), 'Sparse principal component analysis via rotation and truncation', *IEEE Transactions on Neural Networks and Learning Systems* **27**(4), 875–890.

Hyman, J. M. (2005), 'Patch dynamics for multiscale problems', *Computing in Science & Engineering* **7**(3), 47–53.
http://scitation.aip.org/content/aip/journal/cise/7/3/10.1109/MCSE.2005.57

Kevrekidis, I. G., Gear, C. W. & Hummer, G. (2004), 'Equation-free: the computer-assisted analysis of complex, multiscale systems', *A. I. Ch. E. Journal* **50**, 1346–1354.

Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, P. G., Runborg, O. & Theodoropoulos, K. (2003), 'Equation-free, coarse-grained multiscale computation: enabling microscopic simulators to perform system level tasks', *Comm. Math. Sciences* **1**, 715–762.

Kevrekidis, I. G. & Samaey, G. (2009), 'Equation-free multiscale computation: Algorithms and applications', *Annu. Rev. Phys. Chem.* **60**, 321—44.

Leitenmaier, L. & Runborg, O. (2021), Heterogeneous multiscale methods for the landau-lifshitz equation, Technical report, http://arxiv.org/abs/2108.09463.

Liu, P., Samaey, G., Gear, C. W. & Kevrekidis, I. G. (2015), 'On the acceleration of spatially distributed agent-based computations: A patch

dynamics scheme', *Applied Numerical Mathematics* **92**, 54–69. http://www.sciencedirect.com/science/article/pii/S0168927414002086

MacKenzie, T. & Roberts, A. J. (2003), Holistic discretisation of shear dispersion in a two-dimensional channel, *in* K. Burrage & R. B. Sidje, eds, 'Proc. of 10th Computational Techniques and Applications Conference CTAC-2001', Vol. 44, pp. C512–C530.

Maclean, J., Bunder, J. E. & Roberts, A. J. (2020), 'A toolbox of equation-free functions in matlab/octave for efficient system level simulation', *Numerical Algorithms* .

Maclean, J. & Gottwald, G. A. (2015), 'On convergence of higher order schemes for the projective integration method for stiff ordinary differential equations', *Journal of Computational and Applied Mathematics* **288**, 44–69. http://www.sciencedirect.com/science/article/pii/S0377042715002149

Maier, B. & Verfürth, B. (2021), Numerical upscaling for wave equations with time-dependent multiscale coefficients, Technical report, http://arxiv.org/abs/2107.14069.

Marschler, C., Sieber, J., Berkemer, R., Kawamoto, A. & Starke, J. (2014), 'Implicit methods for equation-free analysis: Convergence results and analysis of emergent waves in microscopic traffic models', *SIAM J. Appl. Dyn. Syst.* **13**(2), 1202–1238.

Oron, A., Davis, S. H. & Bankoff, S. G. (1997), 'Long-scale evolution of thin liquid films', *Rev. Mod. Phys.* **69**, 931–980. http://link.aps.org/abstract/RMP/v69/p931.

Petersik, P. (2019–), Equation-free modeling, Technical report, [https://github.com/pjpetersik/eqnfree].

Roberts, A. J. (2003), 'A holistic finite difference approach models linear dynamics consistently', *Mathematics of Computation* **72**, 247–262. http://www.ams.org/mcom/2003-72-241/S0025-5718-02-01448-5

Roberts, A. J. & Kevrekidis, I. G. (2007), 'General tooth boundary conditions for equation free modelling', *SIAM J. Scientific Computing* **29**(4), 1495–1510.

Roberts, A. J., MacKenzie, T. & Bunder, J. (2014), 'A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions', *J. Engineering Mathematics* **86**(1), 175–207. http://arxiv.org/abs/1103.1187

Samaey, G., Kevrekidis, I. G. & Roose, D. (2005), 'The gap-tooth scheme for homogenization problems', *Multiscale Modeling and Simulation* **4**, 278–306.

Samaey, G., Roose, D. & Kevrekidis, I. G. (2006), 'Patch dynamics with buffers for homogenization problems', *J. Comput Phys.* **213**, 264–287.

Sieber, J., Marschler, C. & Starke, J. (2018), 'Convergence of Equation-Free Methods in the Case of Finite Time Scale Separation with Application to Deterministic and Stochastic Systems', *SIAM Journal on Applied Dynamical Systems* **17**(4), 2574–2614.

Taylor, G. I. (1953), 'Dispersion of soluble matter in solvent flowing slowly through a tube', *Proc. Roy. Soc. Lond. A* **219**, 186–203.

Watt, S. D. & Roberts, A. J. (1995), 'The accurate dynamic modelling of contaminant dispersion in channels', *SIAM J. Appl. Math.* **55**(4), 1016–1038. http://epubs.siam.org/sam-bin/dbq/article/25797.

Wilson, S. K., Hunt, R. & Duffy, B. R. (2000), 'The rate of spreading in spin coating', *J. Fluid Mech.* **413**, 65–88.