

Equation-Free function toolbox for Matlab/Octave: Full Developers Manual

A. J. Roberts^{*} John Maclean[†] J. E. Bunder[‡]

March 13, 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	4
2	Projective integration of deterministic ODEs	7
2.1	Introduction	8
2.2	PIRK2(): projective integration of second-order accuracy	11
2.3	egPIMM: Example projective integration of Michaelis–Menton kinetics	18
2.4	PIG(): Projective Integration via a General macroscale integrator	22
2.5	PIRK4(): projective integration of fourth-order accuracy	29
2.6	cdmc(): constraint defined manifold computing	36
2.7	Example: PI using Runge–Kutta macrosolvers	37
2.8	Example: Projective Integration using General macrosolvers	40
2.9	Explore: Projective Integration using constraint-defined manifold computing	42
2.10	To do/discuss	44
3	Patch scheme for given microscale discrete space system	45
3.1	configPatches1(): configures spatial patches in 1D	49
3.2	patchSys1(): interface 1D space to time integrators	60
3.3	patchEdgeInt1(): sets patch-edge values from interpolation over the 1D macroscale	62
3.4	homogenisationExample: simulate heterogeneous diffusion in 1D	69
3.5	homoDiffEdgy1: computational homogenisation of a 1D heterogeneous diffusion by simulation on small patches	74
3.6	BurgersExample: simulate Burgers' PDE on patches	80
3.7	waterWaveExample: simulate a water wave PDE on patches	84
3.8	homoWaveEdgy1: computational homogenisation of a 1D wave by simulation on small patches	90
3.9	waveEdgy1: simulate a 1D, first-order, wave PDE on small patches	95

3.10	<code>Eckhardt2210eg2</code> : example of a 1D heterogeneous diffusion by simulation on small patches	100
3.11	<code>EckhardtEquilib</code> : find an equilibrium of a 1D heterogeneous diffusion via small patches	104
3.12	<code>EckhardtEquilibErrs</code> : explore errors in equilibria of a 1D heterogeneous diffusion on small patches	106
3.13	<code>Eckhardt2210eg1</code> : example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches	111
3.14	<code>homoLanLif1D</code> : computational homogenisation of a 1D heterogeneous Landau–Lifshitz by simulation on small patches	115
3.15	<code>Combescure2022</code> : simulation and continuation of a 1D example nonlinear elasticity, via patches	121
3.16	<code>quasiLogAxes()</code> : transforms plot to quasi-log axes	131
3.17	<code>theRes()</code> : wrapper function to zero for equilibria	134
4	Patches in 2D space	135
4.1	<code>configPatches2()</code> : configures spatial patches in 2D	136
4.2	<code>patchSys2()</code> : interface 2D space to time integrators	150
4.3	<code>patchEdgeInt2()</code> : sets 2D patch edge values from 2D macroscale interpolation	152
4.4	<code>wave2D</code> : example of a wave on patches in 2D	161
4.5	<code>homoDiffEdgy2</code> : computational homogenisation of a 2D diffusion via simulation on small patches	165
4.6	<code>homoDiffSoln2</code> : steady state of a 2D heterogeneous diffusion via small patches	169
4.7	<code>monoscaleDiffEquil2</code> : equilibrium of a 2D monoscale heterogeneous diffusion via small patches	175
4.8	<code>twoscaleDiffEquil2</code> : equilibrium of a 2D twoscale heterogeneous diffusion via small patches	178
4.9	<code>twoscaleDiffEquil2Errs</code> : errors in equilibria of a 2D twoscale heterogeneous diffusion via small patches	181
4.10	<code>abdulleDiffEquil2</code> : equilibrium of a 2D multiscale heterogeneous diffusion via small patches	188
4.11	<code>randAdvecDiffEquil2</code> : equilibrium of a 2D random heterogeneous advection-diffusion via small patches	191
4.12	<code>homoWaveEdgy2</code> : computational homogenisation of a forced, non-autonomous, 2D wave via simulation on small patches	195

5 Patches in 3D space	200
5.1 <code>configPatches3()</code> : configures spatial patches in 3D	201
5.2 <code>patchSys3()</code> : interface 3D space to time integrators	217
5.3 <code>patchEdgeInt3()</code> : sets 3D patch face values from 3D macroscale interpolation	219
5.4 <code>homoDiffEdgy3</code> : computational homogenisation of a 3D diffusion via simulation on small patches	231
5.5 <code>homoDiffBdryEqui13</code> : equilibrium via computational homogenisation of a 3D heterogeneous diffusion on small patches	237
5.6 To do	241
5.7 Miscellaneous tests	242
6 Matlab parallel computation of the patch scheme	255
6.1 <code>chanDispSpmd</code> : simulation of a 1D shear dispersion via simulation on small patches across a channel	257
6.2 <code>rotFilmSpmd</code> : simulation of a 2D shallow water flow on a rotating heterogeneous substrate	265
6.3 <code>homoDiff31spmd</code> : computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion	273
6.4 <code>RK2mesoPatch()</code>	279
6.5 To do	283
A Create, document and test algorithms	284
B Aspects of developing a ‘toolbox’ for patch dynamics	287
B.1 Macroscale grid	288
B.2 Macroscale field variables	289
B.3 Boundary and coupling conditions	290
B.4 Mesotime communication	291
B.5 Projective integration	292
B.6 Lift to many internal modes	293
B.7 Macroscale closure	294
B.8 Exascale fault tolerance	295
B.9 Link to established packages	296

1 Introduction

This Developers Manual contains complete descriptions of the code in each function in the toolbox, and of each example. For concise descriptions of each function, quick start guides, and some basic examples, see the User Manual.

Users Download via <https://github.com/uaa1184615/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](#)), or the staggered patches of 1D water wave equations in `waterWaveExample.m` ([Section 3.7](#)).
- In 2D space adapt the code at the beginning of `configPatches2.m` for nonlinear diffusion ([Section 4.1.1](#)), or the regular patches of the 2D wave PDE of `wave2D.m` ([Section 4.4](#)).
- 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.

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 \(2016b\)](#) 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. [Appendix A](#) outlines some details for contributors.

2 Projective integration of deterministic ODEs

Chapter contents

2.1	Introduction	8
2.2	PIRK2(): projective integration of second-order accuracy	11
2.2.1	Introduction	11
2.2.2	If no arguments, then execute an example	13
2.2.3	The projective integration code	14
2.2.4	If no output specified, then plot the simulation	17
2.3	egPIMM: Example projective integration of Michaelis–Menton kinetics	18
2.4	PIG(): Projective Integration via a General macroscale integrator	22
2.4.1	Introduction	22
2.4.2	If no arguments, then execute an example	24
2.4.3	The projective integration code	26
2.4.4	If no output specified, then plot the simulation	28
2.5	PIRK4(): projective integration of fourth-order accuracy	29
2.5.1	Introduction	29
2.5.2	The projective integration code	31
2.5.3	If no output specified, then plot the simulation	35
2.6	cdmc(): constraint defined manifold computing	36
2.7	Example: PI using Runge–Kutta macrosolvers	37
2.8	Example: Projective Integration using General macrosolvers	40
2.9	Explore: Projective Integration using constraint-defined manifold computing	42
2.10	To do/discuss	44

2.1 Introduction

This section provides some good projective integration functions ([Gear & Kevrekidis 2003b,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.

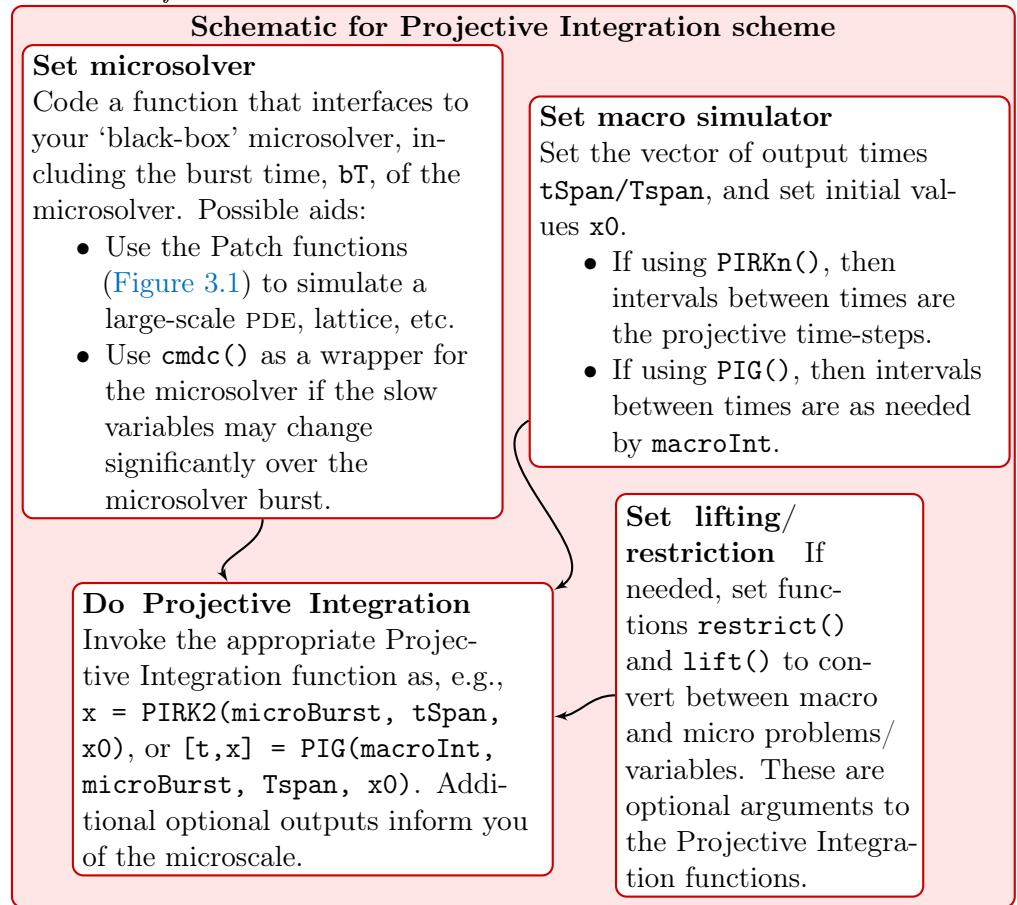
The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

[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. \(2005a\)](#), 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.

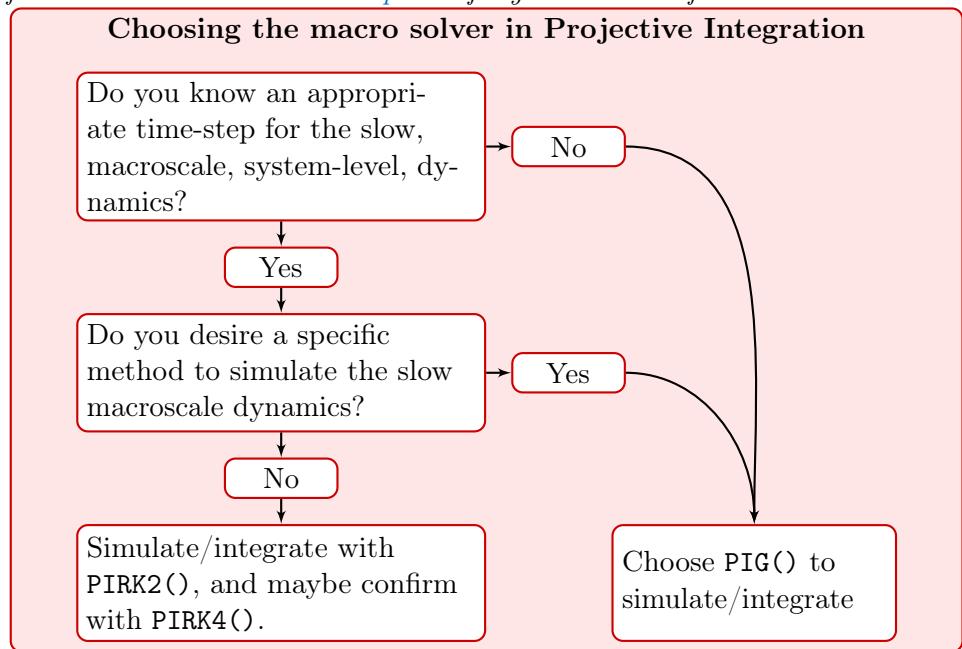
Figure 2.1: The Projective Integration method greatly accelerates simulation/integration of a system exhibiting multiple time scales. The Projective Integration Chapter 2 presents several separate functions, as well as several optional wrapper functions that may be invoked. This chart overviews constructing 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	11
2.2.2	If no arguments, then execute an example	13
2.2.3	The projective integration code	14
2.2.4	If no output specified, then plot the simulation	17

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.

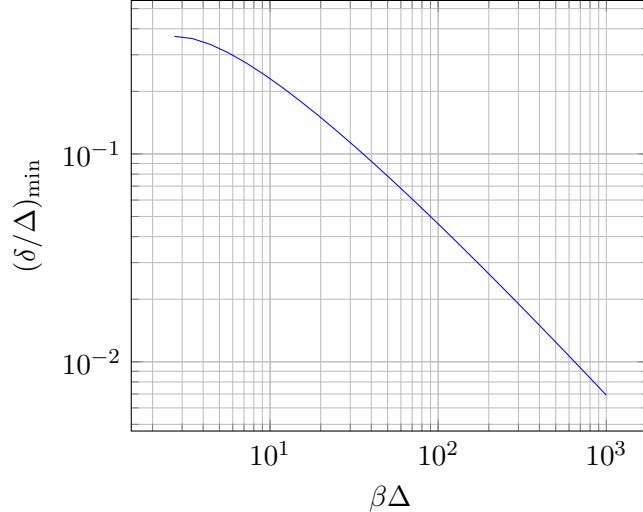
```
[tOut, xOut] = microBurst(tStart, xStart, bT)
```

- 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–Menton 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 `Nans` are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.

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 .



- 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 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, $\Delta = \text{diff}(tSpan)$, 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–Menton enzyme kinetics is expressed as a singularly perturbed system of differential equations for $x(t)$ and $y(t)$:

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

(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 $0 \leq t \leq 6$ 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--Menton 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 $x = \mathbf{x}(1)$ and $y = \mathbf{x}(2)$. 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
25
26 function [ts, xs] = odeOct(dxdt,tSpan,x0)
27     if length(tSpan)>2, ts = tSpan;
28     else ts = linspace(tSpan(1),tSpan(end),21);
29     end
30     % mimic ode45 and ode23, but much slower for non-PI
31     lsode_options('integration method','non-stiff');
32     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
33 end

```

2.2.3 The projective integration code

Determine the number of time-steps and preallocate storage for macroscale estimates.

```

236 nT=length(tSpan);
237 x=nan(nT,length(x0));

```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```

245 nArgOut=nargout();
246 saveMicro = (nArgOut>1);
247 saveFullMicro = (nArgOut>3);
248 saveSvf = (nArgOut>4);

```

Run a preliminary application of the microBurst on the given initial state to help relax to the slow manifold. This is done in addition to the microBurst in the main loop, because the initial state is often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```

261 x0 = reshape(x0,1,[]);
262 [relax_t,relax_x0] = microBurst(tSpan(1),x0,bT);

```

Use the end point of this preliminary microBurst as the initial state for the loop of macro-steps.

```

270 tSpan(1) = relax_t(end);
271 x(1,:)=relax_x0(end,:);

281 if saveMicro
282     tms = cell(nT,1);
283     xms = cell(nT,1);
284     tms{1} = reshape(relax_t,[],1);
285     xms{1} = relax_x0;
286     if saveFullMicro
287         rm.t = cell(nT,1);
288         rm.x = cell(nT,1);
289         if saveSvf
290             svf.t = nan(2*nT-2,1);
291             svf.dx = nan(2*nT-2,length(x0));
292         end
293     end
294 end

```

Loop over the macroscale time-steps Also set an initial rounding tolerance for checking.

```

303 roundingTol = 1e-8;
304 for jT = 2:nT
305     T = tSpan(jT-1);

```

If two applications of the microBurst would cover one entire macroscale time-step, then do so (setting some internal states to NaN); else proceed to projective step.

```

313     if ~isempty(bT) && 2*abs(bT)>=abs(tSpan(jT)-T) && bT*(tSpan(jT)-T)>0
314         [t1,xm1] = microBurst(T, x(jT-1,:), tSpan(jT)-T);
315         x(jT,:) = xm1(end,:);
316         t2 = nan;    xm2 = nan(1,size(xm1,2));
317         dx1 = xm2;  dx2 = xm2;
318     else

```

Run the first application of the microBurst; since this application directly follows from the initial conditions, or from the latest macrostep, this macroscale information is physically meaningful as a simulation of the system. Extract the size of the final time-step.

```
329 [t1,xm1] = microBurst(T, x(jT-1,:), bT);
```

To estimate the derivative by numerical differentiation, we balance approximation error $\|\ddot{x}\|/dt$ with round-off error $\|x\|\epsilon/dt$ by the optimal time-step $dt \approx \sqrt{(\|x\|\epsilon/\|\ddot{x}\|)}$. Omit $\|\ddot{x}\|$ as we do not know it. Also, limit dt to at most the last tenth of the burst, and at least one step.

```

341     nt = length(t1);
342     optdt = min(0.1*(t1(nt)-t1(1)),sqrt(max(rms(xm1))*1e-15));
343     [~,kt] = min(abs(t1(nt)-optdt-t1(1:nt-1)));
344     ktnt = [kt nt];
345     del = t1(nt)-t1(kt);

```

Check for round-off error, and decrease tolerance so that warnings are not repeated unless things get worse.

```

352     xt = [reshape(t1(ktnt),[],1) xm1(ktnt,:)];
353     if norm(diff(xt))/norm(xt,'fro') < roundingTol
354         warning(['significant round-off error in 1st projection at T=' num2str(T)])
355         roundingTol = roundingTol/10;
356     end

```

Find the needed time-step to reach time $tSpan(n+1)$ and form a first estimate $dx1$ of the slow vector field.

```

365     Dt = tSpan(jT)-t1(end);
366     dx1 = (xm1(nt,:)-xm1(kt,:))/del;

```

Project along $dx1$ to form an intermediate approximation of x ; run another application of the microBurst and form a second estimate of the slow vector field (assuming the burst length is the same, or nearly so).

```

376     xint = xm1(end,:)+(Dt-(t1(end)-t1(1)))*dx1;
377     [t2,xm2] = microBurst(T+Dt, xint, bT);

```

As before, choose dt as best we can to estimate derivative.

```

384     nt = length(t2);
385     optdt = min(0.1*(t2(nt)-t2(1)),sqrt(max(rms(xm2))*1e-15));
386     [~,kt] = min(abs(t2(nt)-optdt-t2(1:nt-1)));
387     ktnt = [kt nt];
388     del = t2(nt)-t2(kt);
389     dx2 = (xm2(nt,:)-xm2(kt,:))/del;

```

Check for round-off error, and decrease tolerance so that warnings are not repeated unless things get worse.

```

396     xt = [reshape(t2(ktnt),[],1) xm2(ktnt,:)];
397     if norm(diff(xt))/norm(xt,'fro') < roundingTol
398         warning(['significant round-off error in 2nd projection at T=' num2str(T)])
399         roundingTol = roundingTol/10;
400     end

```

Use the weighted average of the estimates of the slow vector field to take a macro-step.

```

408     x(jT,:) = xm1(end,:)+Dt*(dx1+dx2)/2;

```

Now end the if-statement that tests whether a projective step saves simulation time.

```

416     end

```

If saving trusted microscale data, then populate the cell arrays for the current loop iterate with the time-steps and output of the first application of the microBurst. Separate bursts by NaNs.

```
426     if saveMicro
427         tms{jT} = [reshape(t1,[],1); nan];
428         xms{jT} = [xm1; nan(1,size(xm1,2))];
```

If saving all microscale data, then repeat for the remaining applications of the microBurst.

```
436     if saveFullMicro
437         rm.t{jT} = [reshape(t2,[],1); nan];
438         rm.x{jT} = [xm2; nan(1,size(xm2,2))];
```

If saving Projective Integration estimates of the slow vector field, then populate the corresponding cells with times and estimates.

```
447     if saveSvf
448         svf.t(2*jT-3:2*jT-2) = [t1(end); t2(end)];
449         svf.dx(2*jT-3:2*jT-2,:) = [dx1; dx2];
450     end
451 end
452 end
```

End the main loop over all the macro-steps.

```
458 end
Overwrite x(1,:) with the specified initial condition tSpan(1).
467 x(1,:) = reshape(x0,1,[]);
```

For additional requested output, concatenate all the cells of time and state data into two arrays.

```
475 if saveMicro
476     tms = cell2mat(tms);
477     xms = cell2mat(xms);
478     if saveFullMicro
479         rm.t = cell2mat(rm.t);
480         rm.x = cell2mat(rm.x);
481     end
482 end
```

2.2.4 If no output specified, then plot the simulation

```
490 if nArgOut==0
491     figure, plot(tSpan,x,'o:')
492     title('Projective Simulation with PIRK2')
493 end
```

This concludes PIRK2().

```
500 end
```

2.3 egPIMM: Example projective integration of Michaelis–Menton kinetics

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

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

(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.

```
31 clear all, close all
32 global MMepsilon
33 MMepsilon = 0.1
```

First, the end of this section encodes the computation of bursts of the Michaelis–Menton 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–Menton system, at time $t = 0$, of $(x, y) = (1, 0)$ (off the slow manifold).

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

[Figure 2.4](#) plots the macroscale results showing the long time decay of the Michaelis–Menton 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.

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

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

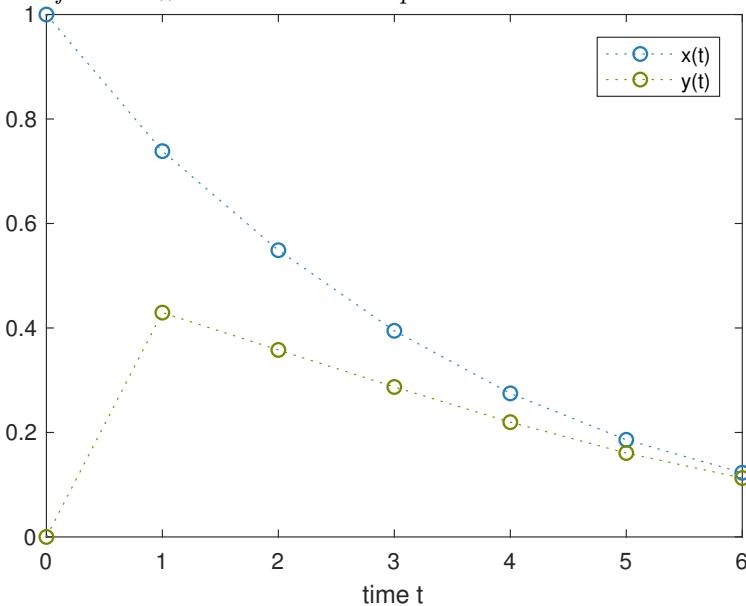


Figure 2.5 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable $x(t)$ is also affected by the initial transient (hence other schemes which ‘freeze’ slow variables are less accurate).

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 2003a, 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 .

```

115 ts = 0:-1:-5
116 [xs,tMicro,xMicro] = PIRK2(@MMburst, ts, 0.2*[1;1], 3*MMepsilon);
117 figure, plot(ts, xs, 'o:', tMicro, xMicro)
118 xlabel('time t'), legend('x(t)', 'y(t)')
119 title('backward integration showing points with bursts')
120 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 $x = \mathbf{x}(1)$ and $y = \mathbf{x}(2)$. 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) )];

```

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.

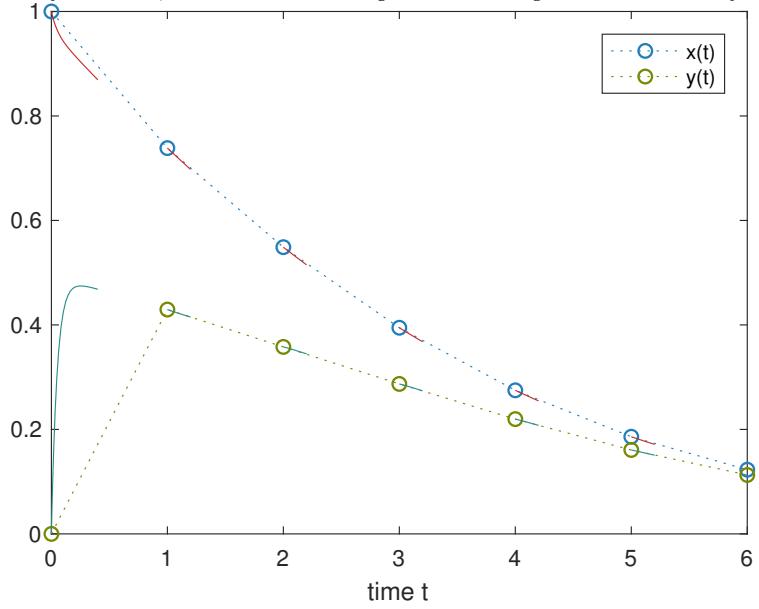
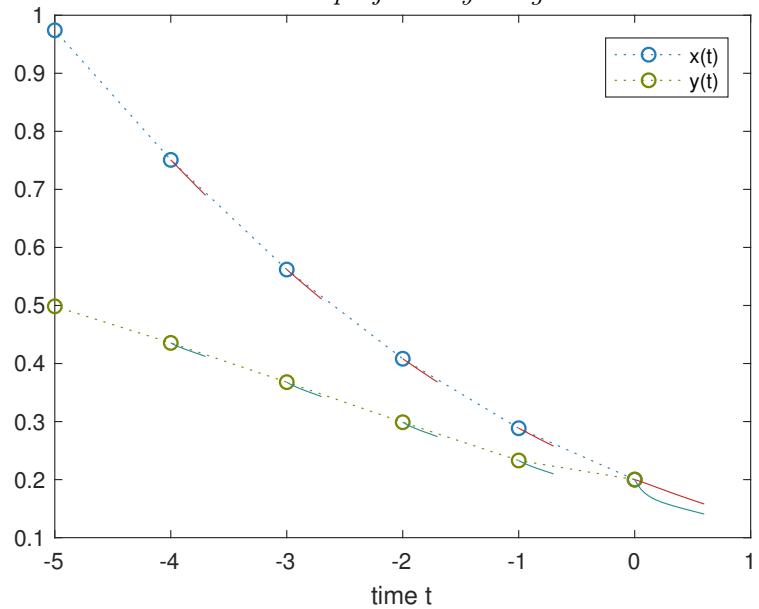


Figure 2.6: Michaelis–Menten enzyme kinetics at $\epsilon = 0.1$ simulated backward with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to projectively integrate backward in time.



```
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
```

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

Section contents

2.4.1	Introduction	22
2.4.2	If no arguments, then execute an example	24
2.4.3	The projective integration code	26
2.4.4	If no output specified, then plot the simulation	28

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. \(2005b\)](#), 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

$$[Ts, Xs] = \text{macroInt}(F, Tspan, X0)$$

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 i th *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.

- \mathbf{X} , an $L \times N$ array of the computed solution: the i th row of \mathbf{X} , $\mathbf{X}(i, :)$, 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(...)`

- \mathbf{tms} , optional, is an ℓ -dimensional column vector containing microscale times with bursts, each burst separated by `NaN`;
- \mathbf{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

- \mathbf{svf} , optional, a struct containing the Projective Integration estimates of the slow vector field.
 - $\mathbf{svf}.T$ 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()`.
 - $\mathbf{svf}.dX$ 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

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

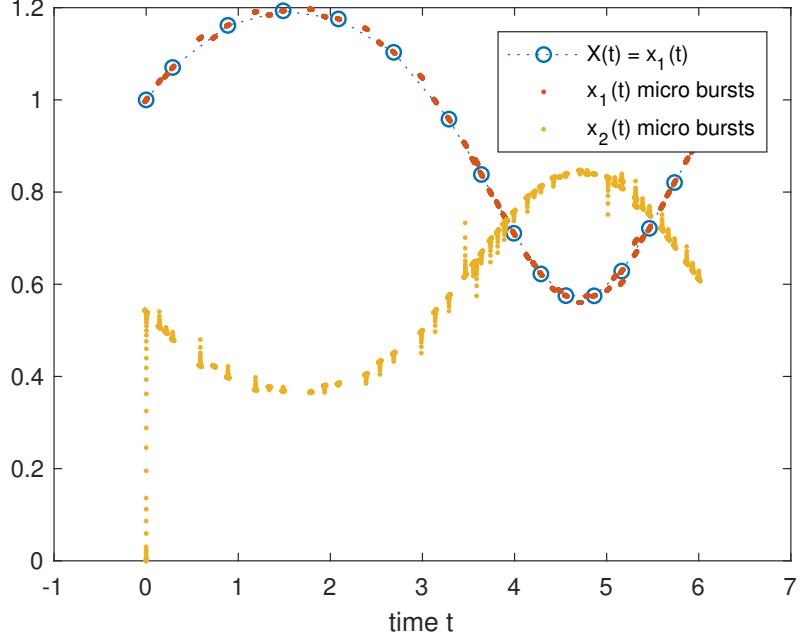
The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and plots a solution of the system (2.1) over time $0 \leq t \leq 6$ for parameter $\epsilon = 10^{-3}$ (Figure 2.7). Whenever needed by `microBurst()`, the microscale 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.

```
214 epsilon = 1e-3;
215 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
216           ( 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 $\Delta \leq 1$ and then double the usual formula for safety.

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 $(x_1(t), x_2(t)) = (\text{red}, \text{yellow})$ dots.



```

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

```

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

```

237 restrict = @(x) x(1);
238 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 $0 \leq t \leq 6$ 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 2003a, Frewen et al. 2009).

```

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

```

Plot output of this projective integration.

```

260 figure, plot(Ts,Xs,'o:',tms,xms,'.')
261 title('Projective integration of singularly perturbed ODE')
262 xlabel('time t')
263 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.

```
269 return
270 end%if no arguments
```

2.4.3 The projective integration code

If no lifting/restriction functions are provided, then assign them to be the identity functions.

```
287 if nargin < 5 || isempty(restrict)
288     lift=@(X,xApprox) X;
289     restrict=@(x) x;
290 end
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
298 nArgOut = nargin();
299 saveMicro = (nArgOut>2);
300 saveSvf = (nArgOut>4);
```

Find the number of time-steps at which output is expected, and the number of variables.

```
308 nT = length(Tspan)-1;
309 nx = length(x0);
310 nX = length(restrict(x0));
```

Reformulate the microsolver to use `cdmc()`, unless flagged otherwise. The result is that the solution from microBurst will terminate at the given initial time.

```
320 if nargin<7
321     microBurst = @(t,x) cdmc(microBurst,t,x);
322 else
323     warning(['A ' class(cdmFlag) ' seventh input to PIG()'...
324         ' PIG will not use constraint-defined manifold computing.'])
325 end
```

Execute a preliminary application of the microBurst on the initial state. This is done in addition to the microBurst in the main loop, because the initial state is often far from the attracting slow manifold.

```
337 [relaxT,x0MicroRelax] = microBurst(Tspan(1),x0);
338 xMicroLast = x0MicroRelax(end,:);
339 X0Relax = restrict(xMicroLast);
```

Update the initial time.

```
346 Tspan(1) = relaxT(end);
```

Allocate cell arrays for times and states for any of the outputs requested by the user. If saving information, then record the first application of the microBurst. It is unknown a priori how many applications of microBurst will

be required; this code may be run more efficiently if the correct number is used in place of $nT+1$ as the dimension of the cell arrays.

```

358 if saveMicro
359     tms=cell(nT+1,1); xms=cell(nT+1,1);
360     n=1;
361     tms{n} = reshape(relaxT,[],1);
362     xms{n} = x0MicroRelax;
363
364     if saveSvf
365         svf.T = cell(nT+1,1);
366         svf.dX = cell(nT+1,1);
367     else
368         svf = [];
369     end
370 else
371     tms = [] ; xms = [] ; svf = [] ;
372 end

```

Define a function of macro simulation The idea of `PIG()` is to use the output from the `microBurst()` to approximate an unknown function $F(t, X)$ that computes $d\vec{X}/dt$. This approximation is then used in the system/user-defined ‘coarse solver’ `macroInt()`. The approximation is computed in the function

```
385 function [dXdt]=PIFun(t,X)
```

Run a microBurst from the given macroscale initial values.

```

391 x = lift(X,xMicroLast);
392 [tTmp,xMicroTmp] = microBurst(t,reshape(x,[],1));
393 xMicroLast = xMicroTmp(end,:).';

```

Compute the standard Projective Integration approximation of the slow vector field.

```

400 X2 = restrict(xMicroTmp(end,:));
401 X1 = restrict(xMicroTmp(end-1,:));
402 dt = tTmp(end)-tTmp(end-1);
403 dXdt = (X2 - X1).'/dt;

```

Save the microscale data, and the Projective Integration slow vector field, if requested.

```

410 if saveMicro
411     n=n+1;
412     tms{n} = [reshape(tTmp,[],1); nan];
413     xms{n} = [xMicroTmp; nan(1,nx)];
414     if saveSvf
415         svf.T{n-1} = t;
416         svf.dX{n-1} = dXdt;
417     end

```

```

418     end
419 end% PIFun function

```

Invoke the macroscale integration Integrate PIF() with the user-specified simulator macroInt(). For some reason, in MATLAB/Octave we need to use a one-line function, PIF, that invokes the above macroscale function, PIFun. We also need to use feval because macroInt() has multiple outputs.

```

432 PIF = @(t,x) PIFun(t,x);
433 [T,X] = feval(macroInt,PIF,Tspan,X0Relax.');

```

Overwrite X(1,:) and T(1), which a user expects to be X0 and Tspan(1) respectively, with the given initial conditions.

```

442 X(1,:) = restrict(x0);
443 T(1) = Tspan(1);

```

Concatenate all the additional requested outputs into arrays.

```

450 if saveMicro
451     tms = cell2mat(tms);
452     xms = cell2mat(xms);
453     if saveSvf
454         svf.T = cell2mat(svf.T);
455         svf.dX = cell2mat(svf.dX);
456     end
457 end

```

2.4.4 If no output specified, then plot the simulation

```

465 if nArgOut==0
466     figure, plot(T,X,'o:')
467     title('Projective Simulation via PIG')
468 end

```

This concludes PIG().

```

476 end

```

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

Section contents

2.5.1	Introduction	29
2.5.2	The projective integration code	31
2.5.3	If no output specified, then plot the simulation	35

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

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

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 $-5 \leq t \leq 0$ 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--Menton')
58
59 % Plot the solution
60 figure, plot(ts,x)
61 xlabel('time t'), ylabel('solution')
62 title('Backwards-time projective integration of Michaelis--Menton')
63
64 end%if no arguments
```

Upon finishing execution of the example, exit this function.

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 $x = \mathbf{x}(1)$ and $y = \mathbf{x}(2)$. 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
25
26 function [ts, xs] = odeOct(dxdt,tSpan,x0)
27     if length(tSpan)>2, ts = tSpan;
28     else ts = linspace(tSpan(1),tSpan(end),21);
29     end
30     % mimic ode45 and ode23, but much slower for non-PI
31     lsode_options('integration method','non-stiff');
32     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
33 end

```

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.

- `tSpan` is an ℓ -vector of times at which the user requests output, of which the first element is always the initial time. `PIRK4()` 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 `Nans` 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 slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a burst.

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

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 = PIRK4(microBurst,tSpan,x0,bT)`.

However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK4()` 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 4ℓ 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 $4\ell \times n$ array containing the estimated slow vector field.

2.5.2 The projective integration code

Determine the number of time-steps and preallocate storage for macroscale estimates.

```
194 nT = length(tSpan);
195 x = nan(nT,length(x0));
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
203 nArgOut = nargin();
204 saveMicro = (nArgOut>1);
205 saveFullMicro = (nArgOut>3);
206 saveSvf = (nArgOut>4);
```

Run a preliminary application of the micro-burst on the initial state to help relax to the slow manifold. This is done in addition to the micro-burst in the main loop, because the initial state is often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```
219 x0 = reshape(x0,1,[]);
220 [relax_t,relax_x0] = microBurst(tSpan(1),x0,bT);
```

Use the end point of the micro-burst as the initial state for the macroscale time-steps.

```
228 tSpan(1) = relax_t(end);
229 x(1,:) = relax_x0(end,:);
```

If saving information, then record the first application of the micro-burst. Allocate cell arrays for times and states for outputs requested by the user, as concatenating cells is much faster than iteratively extending arrays.

```
239 if saveMicro
240     tms = cell(nT,1);
241     xms = cell(nT,1);
242     tms{1} = reshape(relax_t,[],1);
243     xms{1} = relax_x0;
244     if saveFullMicro
245         rm.t = cell(nT,1);
246         rm.x = cell(nT,1);
247         if saveSvf
248             svf.t = nan(4*nT-4,1);
249             svf.dx = nan(4*nT-4,length(x0));
250         end
251     end
252 end
```

Loop over the macroscale time-steps

```
260 for jT = 2:nT
261     T = tSpan(jT-1);
```

If four applications of the micro-burst would cover the entire macroscale time-step, then do so (setting some internal states to NaN); else proceed to projective step.

```
270 if ~isempty(bT) && 4*abs(bT)>=abs(tSpan(jT)-T) && bT*(tSpan(jT)-T)>0
271     [t1,xm1] = microBurst(T, x(jT-1,:), tSpan(jT)-T);
272     x(jT,:) = xm1(end,:);
```

```

273         t2=nan; xm2=nan(1,size(xm1,2));
274         t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
275     else

```

Run the first application of the micro-burst; since this application directly follows from the initial conditions, or from the latest macrostep, this microscale information is physically meaningful as a simulation of the system. Extract the size of the final time-step.

```

286     [t1,xm1] = microBurst(T, x(jT-1,:), bT);
287     del = t1(end)-t1(end-1);

```

Check for round-off error.

```

293     xt = [reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
294     roundingTol = 1e-8;
295     if norm(diff(xt))/norm(xt,'fro') < roundingTol
296     warning(['significant round-off error in 1st projection at T=' num2str(T)
297     end

```

Find the needed time-step to reach time $tSpan(n+1)$ and form a first estimate $dx1$ of the slow vector field.

```

306     Dt = tSpan(jT)-t1(end);
307     dx1 = (xm1(end,:)-xm1(end-1,:))/del;

```

Assume burst times are the same length for this macro-step, or effectively so (recall that bT may be empty as it may be only coded and known in `microBurst()`).

```
316     abT = t1(end)-t1(1);
```

Project along $dx1$ to form an intermediate approximation of x ; run another application of the micro-burst and form a second estimate of the slow vector field.

```

327     xint = xm1(end,:)+(Dt/2-abT)*dx1;
328     [t2,xm2] = microBurst(T+Dt/2, xint, bT);
329     del = t2(end)-t2(end-1);
330     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
331
332     xint = xm1(end,:)+(Dt/2-abT)*dx2;
333     [t3,xm3] = microBurst(T+Dt/2, xint, bT);
334     del = t3(end)-t3(end-1);
335     dx3 = (xm3(end,:)-xm3(end-1,:))/del;
336
337     xint = xm1(end,:)+(Dt-abT)*dx3;
338     [t4,xm4] = microBurst(T+Dt, xint, bT);
339     del = t4(end)-t4(end-1);
340     dx4 = (xm4(end,:)-xm4(end-1,:))/del;

```

Check for round-off error.

```

346     xt = [reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
347     if norm(diff(xt))/norm(xt,'fro') < roundingTol

```

```

348     warning(['significant round-off error in 2nd projection at T=' num2str(T))
349     end

```

Use the weighted average of the estimates of the slow vector field to take a macrostep.

```

357     x(jT,:) = xm1(end,:)+Dt*(dx1+2*dx2+2*dx3+dx4)/6;

```

Now end the if-statement that tests whether a projective step saves simulation time.

```

365     end

```

If saving trusted microscale data, then populate the cell arrays for the current loop iterate with the time-steps and output of the first application of the micro-burst. Separate bursts by NaNs.

```

375     if saveMicro
376         tms{jT} = [reshape(t1,[],1); nan];
377         xms{jT} = [xm1; nan(1,size(xm1,2))];

```

If saving all microscale data, then repeat for the remaining applications of the micro-burst.

```

385     if saveFullMicro
386         rm.t{jT} = [reshape(t2,[],1); nan;...
387                     reshape(t3,[],1); nan;...
388                     reshape(t4,[],1); nan];
389         rm.x{jT} = [xm2; nan(1,size(xm2,2));...
390                     xm3; nan(1,size(xm2,2));...
391                     xm4; nan(1,size(xm2,2))];

```

If saving Projective Integration estimates of the slow vector field, then populate the corresponding cells with times and estimates.

```

400     if saveSvf
401         svf.t(4*jT-7:4*jT-4) = [t1(end); t2(end); t3(end); t4(end)];
402         svf.dx(4*jT-7:4*jT-4,:) = [dx1; dx2; dx3; dx4];
403     end
404     end
405 end

```

End of the main loop of all macro-steps.

```

411 end

```

Overwrite $x(1,:)$ with the specified initial state $tSpan(1)$.

```

420 x(1,:) = reshape(x0,1,[]);

```

For additional requested output, concatenate all the cells of time and state data into two arrays.

```

428 if saveMicro
429     tms = cell2mat(tms);
430     xms = cell2mat(xms);
431     if saveFullMicro

```

```
432         rm.t = cell2mat(rm.t);  
433         rm.x = cell2mat(rm.x);  
434     end  
435 end
```

2.5.3 If no output specified, then plot the simulation

```
443 if nArgOut==0  
444     figure, plot(tSpan,x,'o:')
```

```
445     title('Projective Simulation with PIRK4')
```

```
446 end
```

This concludes PIRK4().

```
453 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.

Begin with a standard application of the micro-burst. Need `feval` as `microBurst` has multiple outputs.

```
56 [t1,x1] = feval(microBurst,t0,x0);
57 bT = t1(end)-t1(1);
```

Project backwards to before the initial time, then simulate just one burst forward to obtain a simulation burst that ends at the original `t0`.

```
66 dxdt = (x1(end,:) - x1(end-1,:))/(t1(end) - t1(end-1));
67 x0 = x1(end,:)-2*bT*dxdt;
68 t0 = t1(1)-bT;
69 [t2,x2] = feval(microBurst,t0,x0.');
```

Return both sets of output(?), although only `(t2,x2)` should be used in Projective Integration—maybe safer to return only `(t2,x2)`.

```
77 ts = [t1(:); t2(:)];
78 xs = [x1; x2];
```

2.7 Example: PI using Runge–Kutta macrosolvers

This script demonstrates the PIRK4() scheme that uses a Runge–Kutta macrosolver, applied to simple linear systems with some slow and fast directions.

Clear workspace and set a seed.

```
15 clear
16 rand('seed',1) % albeit discouraged in Matlab
17 global dxdt
```

The majority of this example involves setting up details for the microsolver. We use a simple function gen_linear_system() that outputs a function $f(t, x) = A\vec{x} + \vec{b}$, where matrix A has some eigenvalues with large negative real part, corresponding to fast variables, and some eigenvalues with real part close to zero, corresponding to slow variables. The function gen_linear_system() requires that we specify bounds on the real part of the strongly stable eigenvalues,

```
32 fastband = [-5e2; -1e2];
```

and bounds on the real part of the weakly stable/unstable eigenvalues,

```
39 slowband = [-0.002; 0.002];
```

We now generate a random linear system with seven fast and three slow variables.

```
46 dxdt = gen_linear_system(7,3,fastband,slowband);
```

Set the macroscale times at which we request output from the PI scheme and the initial state.

```
56 tSpan = 0:1:20;
57 x0 = linspace(-10,10,10)';
```

We implement the PI scheme, saving the coarse states in \mathbf{x} , the ‘trusted’ applications of the microsolver in \mathbf{tms} and \mathbf{xms} , and the additional applications of the microsolver in \mathbf{rm} (the second, third and fourth outputs are optional).

```
70 [x, tms, xms, rm] = PIRK4(@linearBurst, tSpan, x0);
```

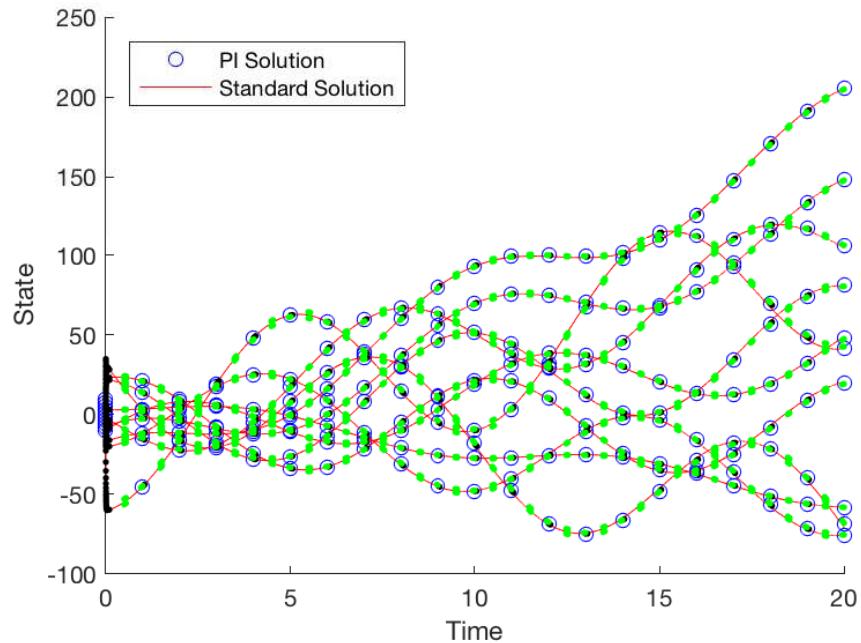
To verify, we also compute the trajectories using a standard integrator.

```
77 if ~exist('OCTAVE_VERSION','builtin')
78     [tt,xode] = ode45(dxdt,tSpan([1,end]),x0);
79 else % octave version
80     tt = linspace(tSpan(1),tSpan(end),101);
81     xode = lsode(@(x,t) dxdt(t,x),x0,tt);
82 end
```

[Figure 2.8](#) plots the output.

```
98 clf()
99 hold on
100 PI_sol=plot(tSpan,x,'bo');
101 std_sol=plot(tt,xode,'r');
```

Figure 2.8: Demonstration of PIRK4(). From initial conditions, the system rapidly transitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.



```

102 plot(tms,xms,'k.', rm.t,rm.x,'g.');
103 legend([PI_sol(1),std_sol(1)],'PI Solution',...
104     'Standard Solution','Location','NorthWest')
105 xlabel('Time'), ylabel('State')

Save plot to a file.

111 if ~exist('OCTAVE_VERSION','builtin')
112 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
113 print('-depsc2','PIRKexample')
114 end

```

A micro-burst simulation Used by `PIRKexample.m`. Code the micro-burst function using simple Euler steps. As a rule of thumb, the time-steps dt should satisfy $dt \leq 1/|\text{fastband}(1)|$ and the time to simulate with each application of the microsolver, bT , should be larger than or equal to $1/|\text{fastband}(2)|$. We set the integration scheme to be used in the microsolver. Since the time-steps are so small, we just use the forward Euler scheme

```

17 function [ts, xs] = linearBurst(ti, xi, varargin)
18 global dxdt
19 dt = 0.001;
20 ts = ti+(0:dt:0.05)';
21 nts = length(ts);
22 xs = NaN(nts,length(xi));

```

```
23 xs(1,:)=xi;
24 for k=2:nts
25     xi = xi + dt*dxdt(ts(k),xi.')';
26     xs(k,:)=xi;
27 end
28 end
```

2.8 Example: Projective Integration using General macrosolvers

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation. The aim is to use a standard non-stiff numerical integrator, such as `ode45()`, on the slow, long-time macroscale. For this stiff system, `PIG()` is an order of magnitude faster than ordinary use of `ode45`.

```
18 clear all, close all
```

Set time scale separation and the underlying ODES:

$$\frac{dx_1}{dt} = \cos x_1 \sin x_2 \cos t, \quad \frac{dx_2}{dt} = \frac{1}{\epsilon}(-x_2 + \cos x_1).$$

```
30 epsilon = 1e-4;
31 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
32 (cos(x(1))-x(2))/epsilon ];
```

Set the ‘black-box’ microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

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

Set initial conditions, and the time to be covered by the macrosolver.

```
52 x0 = [1 0.9];
53 tSpan = [0 5];
```

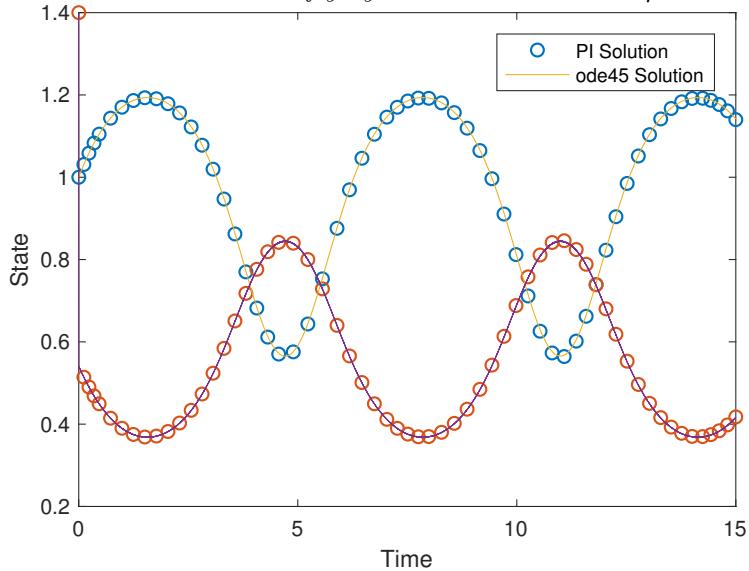
Now time and integrate the above system over `tSpan` using `PIG()` and, for comparison, a brute force implementation of `ode45/lsode`. Report the time taken by each method (in seconds).

```
62 if ~exist('OCTAVE_VERSION','builtin')
63     macInt='ode45'; else macInt='odeOct'; end
64 tic
65 [ts,xs,tms,xms] = PIG(macInt,microBurst,tSpan,x0);
66 secsPIGusingODEasMacro = toc
67 tic
68 [tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
69 secsODEalone = toc
```

Plot the output on two figures, showing the truth and macrosteps on both, and all applications of the microsolver on the first figure.

```
79 figure
80 h = plot(ts,xs,'o', tClassic,xClassic,'-', tms,xms,'.');
81 legend(h(1:2:5),'Pro Int method','classic method','PI microsolver')
82 xlabel('Time'), ylabel('State')
83
84 figure
85 h = plot(ts,xs,'o', tClassic,xClassic,'-');
```

Figure 2.9: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver `ode45/lode`.



```

86 legend(h([1 3]),'Pro Int method','classic method')
87 xlabel('Time'), ylabel('State')
88 if ~exist('OCTAVE_VERSION','builtin')
89 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
90 %print('-depsc2','PIGExample')
91 end

```

Figure 2.9 plots the output.

- The problem may be made more stiff or less stiff by changing the time-scale separation parameter $\epsilon = \text{epsilon}$. The compute time of `PIG()` is almost independent of ϵ , whereas that of `ode45()` is proportional to $1/\epsilon$.

If the problem is ‘semi-stiff’ (larger ϵ), then `PIG()`’s default of using `cdmc()` avoids nonsense ([Section 2.9](#)).

- The stiff but low dimensional problem in this example may be solved efficiently by a standard stiff solver (e.g., `ode15s()`). The real advantage of the Projective Integration schemes is in high dimensional stiff problems, that are not efficiently solved by most standard methods.

2.9 Explore: Projective Integration using constraint-defined manifold computing

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation in which the time scale separation is not large. The results demonstrate the value of the default `cdmc()` wrapper for the microsolver.

```
16 clear all, close all
```

Set a weak time scale separation, and the underlying ODES:

$$\frac{dx_1}{dt} = \cos x_1 \sin x_2 \cos t, \quad \frac{dx_2}{dt} = \frac{1}{\epsilon}(-x_2 + \cos x_1).$$

```
28 epsilon = 0.01;
29 dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
30             (cos(x(1))-x(2))/epsilon ];
```

Set the microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

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

Set initial conditions, and the time to be covered by the macrosolver.

```
50 x0 = [1 0];
51 tSpan=0:0.5:15;
```

Simulate using `PIG()`, first without the default treatment of `cdmc` for the microsolver and second with. Generate a trusted solution using standard numerical methods.

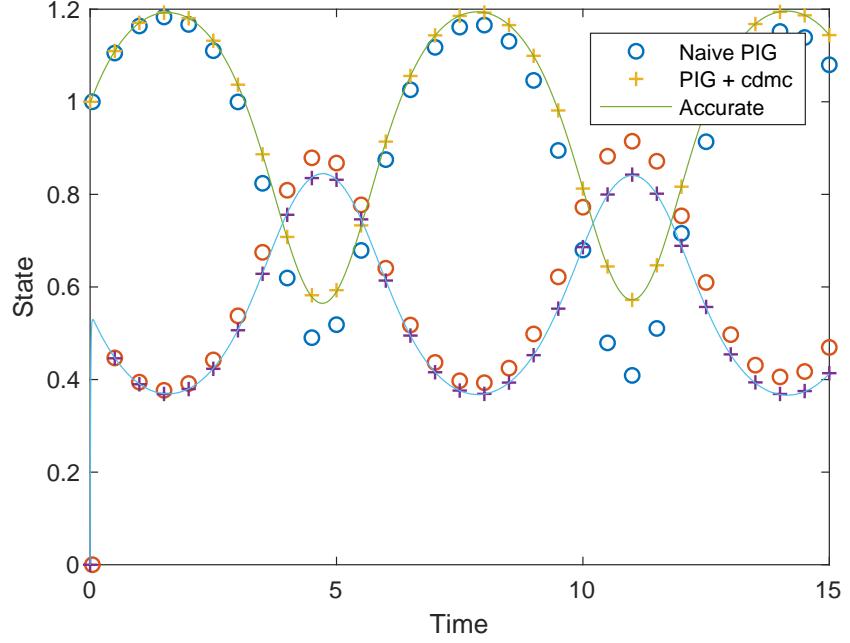
```
62 if ~exist('OCTAVE_VERSION','builtin')
63     macInt='ode45'; else macInt='odeOct'; end
64 [nt,nx] = PIG(macInt,microBurst,tSpan,x0,[],[],'no cdmc');
65 [ct,cx] = PIG(macInt,microBurst,tSpan,x0);
66 [tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
```

[Figure 2.10](#) plots the output.

```
83 figure
84 h = plot(nt,nx,'rx', ct,cx,'bo', tClassic,xClassic,'-');
85 legend(h(1:2:5),'Naive PIG','PIG + cdmc','Accurate')
86 xlabel('Time'), ylabel('State')
87 if ~exist('OCTAVE_VERSION','builtin')
88 set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
89 %print('-depsc2','PIGExplore')
90 end
```

A source of error in the standard `PIG()` scheme is the finite length of each burst, `bT`. This computes a time derivative at a time that is significantly different to that requested by standard coded schemes. Set `bT` to `20*epsilon`

Figure 2.10: Accurate simulation of a weakly stiff non-autonomous system by PIG() using cdm(), and an inaccurate solution using a naive application of PIG().



or `50*epsilon`¹ to worsen the error in both schemes. This example reflects a general principle: most Projective Integration schemes incur a global error term proportional to the burst time of the microsolver and independent of the order of the microsolver. The PIRKn() schemes are written to eliminate this error, but PIG() works with any user-defined macrosolver and cannot reduce this error, except by using the function `cdm()`, its default.

¹ This example is quite extreme: at `bT=50*epsilon`, it would be computationally much cheaper to simulate the entire length of `tSpan` using the microsolver alone.

2.10 To do/discuss

- Implement lifting and restriction for PIRK_n() functions.
- Could implement Projective Integration by ‘arbitrary’ Runge–Kutta scheme; that is, by having the user input a particular Butcher table—surely only specialists would be interested.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the ‘Events’ function handle in ode23.
- Need projective integration of systems with fast oscillations, perhaps by DMD.
- Need projective integration for stochastic systems.

3 Patch scheme for given microscale discrete space system

Chapter contents

3.1	<code>configPatches1()</code> : configures spatial patches in 1D	49
3.1.1	If no arguments, then execute an example	52
3.1.2	Parse input arguments and defaults	54
3.1.3	The code to make patches and interpolation	55
3.1.4	Set ensemble inter-patch communication	57
3.2	<code>patchSys1()</code> : interface 1D space to time integrators	60
3.3	<code>patchEdgeInt1()</code> : sets patch-edge values from interpolation over the 1D macroscale	62
3.3.1	Periodic macroscale interpolation schemes	63
3.3.2	Non-periodic macroscale interpolation	66
3.4	<code>homogenisationExample</code> : simulate heterogeneous diffusion in 1D	69
3.4.1	Script to simulate via stiff or projective integration	70
3.4.2	<code>heteroDiff()</code> : heterogeneous diffusion	72
3.4.3	<code>heteroBurst()</code> : a burst of heterogeneous diffusion	73
3.5	<code>homoDiffEdgy1</code> : computational homogenisation of a 1D heterogeneous diffusion by simulation on small patches	74
3.5.1	Script code to simulate heterogeneous diffusion systems	74
3.5.2	<code>heteroDiff()</code> : heterogeneous diffusion	78
3.6	<code>BurgersExample</code> : simulate Burgers' PDE on patches	80
3.6.1	Script code to simulate a microscale space-time map	80
3.6.2	Alternatively use projective integration	81
3.6.3	<code>burgersMap()</code> : discretise the PDE microscale	83
3.6.4	<code>burgerBurst()</code> : code a burst of the patch map	83
3.7	<code>waterWaveExample</code> : simulate a water wave PDE on patches	84
3.7.1	Script code to simulate wave systems	85
3.7.2	<code>idealWavePDE()</code> : ideal wave PDE	87
3.7.3	<code>waterWavePDE()</code> : water wave PDE	88

3.8	<code>homoWaveEdgy1</code> : computational homogenisation of a 1D wave by simulation on small patches	90
3.8.1	Script code to simulate heterogeneous wave systems	90
3.8.2	<code>heteroWave()</code> : wave in heterogeneous media with weak viscous damping	93
3.9	<code>waveEdgy1</code> : simulate a 1D, first-order, wave PDE on small patches	95
3.9.1	Script code to simulate heterogeneous wave systems	96
3.9.2	<code>waveFirst()</code> : first-order wave PDE	99
3.10	<code>Eckhardt2210eg2</code> : example of a 1D heterogeneous diffusion by simulation on small patches	100
3.10.1	Simulate heterogeneous diffusion systems	101
3.10.2	<code>heteroDiffF()</code> : forced heterogeneous diffusion	103
3.11	<code>EckhardtEquilib</code> : find an equilibrium of a 1D heterogeneous diffusion via small patches	104
3.12	<code>EckhardtEquilibErrs</code> : explore errors in equilibria of a 1D heterogeneous diffusion on small patches	106
3.13	<code>Eckhardt2210eg1</code> : example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches	111
3.13.1	Simulate heterogeneous diffusion systems	112
3.13.2	<code>heteroBurstF()</code> : a burst of heterogeneous diffusion	114
3.14	<code>homoLanLif1D</code> : computational homogenisation of a 1D heterogeneous Landau–Lifshitz by simulation on small patches	115
3.14.1	Script code to simulate heterogeneous diffusion systems	116
3.14.2	Spectrum of the coded patch system	118
3.14.3	<code>heteroLanLif1D()</code> : heterogeneous Landau–Lifshitz PDE	120
3.15	<code>Combescure2022</code> : simulation and continuation of a 1D example nonlinear elasticity, via patches	121
3.15.1	Configure heterogeneous toy elasticity systems	123
3.15.2	Simulate in time	125
3.15.3	MatCont continuation	126
3.15.4	<code>matContSys</code> : basic function for MatCont analysis	128
3.15.5	<code>dudtSys()</code> : wraps around the patch wrapper	128
3.15.6	<code>heteroNLE()</code> : forced heterogeneous elasticity	129
3.16	<code>quasiLogAxes()</code> : transforms plot to quasi-log axes	131

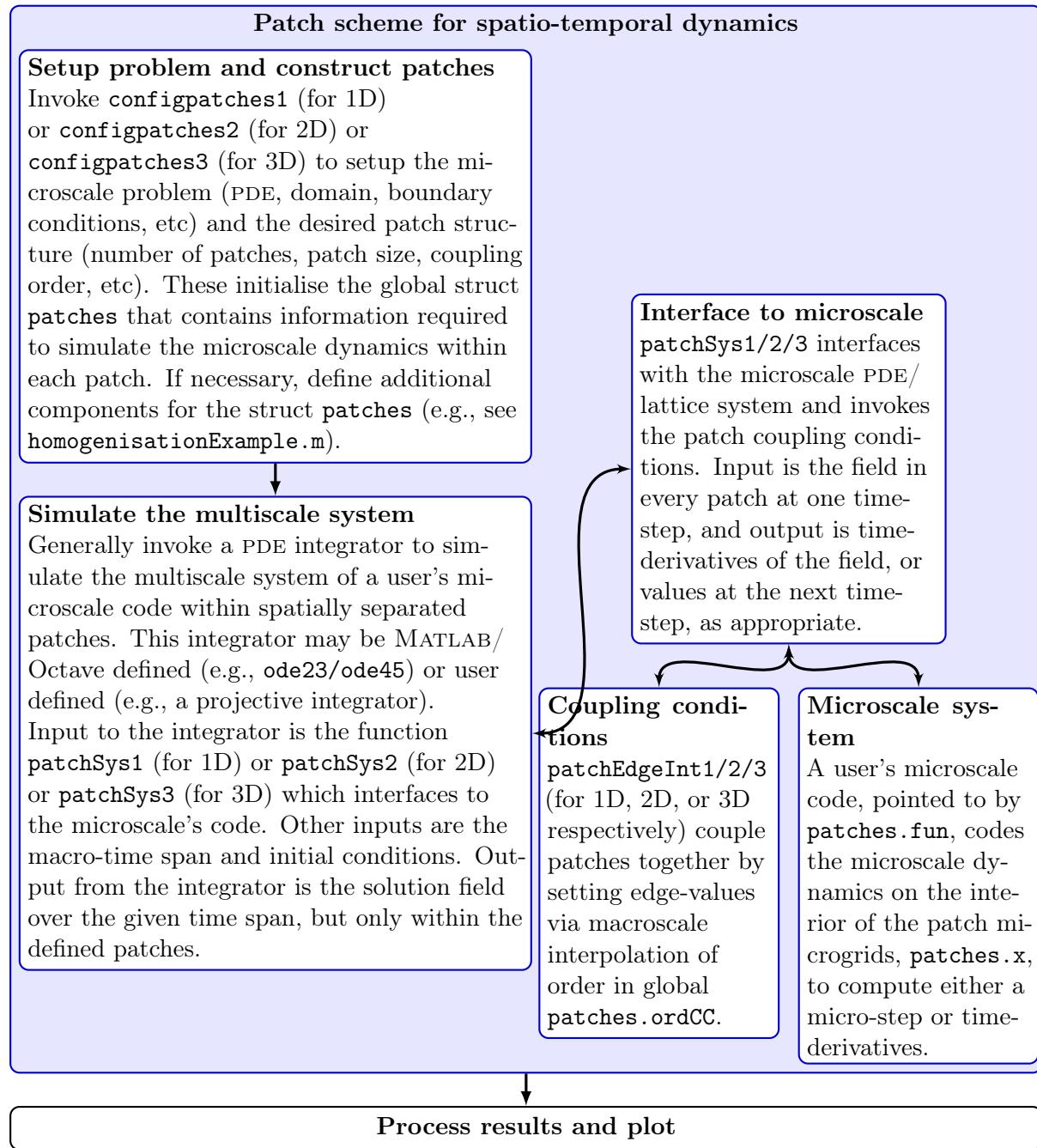
3.17 theRes(): wrapper function to zero for equilibria 134

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 2016b).

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 PDEs, 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(): configures spatial patches in 1D

Section contents

3.1.1	If no arguments, then execute an example	52
3.1.2	Parse input arguments and defaults	54
3.1.3	The code to make patches and interpolation	55
3.1.4	Set ensemble inter-patch communication	57

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)
```

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 if 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. So either $\text{ratio} = \frac{1}{2}$ 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 must be odd so that there is a centre-patch lattice point.
- `nEdge` (not yet implemented), *optional*, default=1, for each patch, 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.
 - 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

- ```
181 if nargout==0, global patches, end
```
- `.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×1×1×nPatch` array of the regular spatial locations  $x_{iI}$  of the  $i$ th microscale grid point in the  $I$ th 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.

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

- `.le`, `.ri` determine inter-patch coupling of members in an ensemble.  
Each a column vector of length `nEnsem`.
- `.cs` either
  - [] 0D, or
  - if `nEnsem` = 1,  $(nSubP(1) - 1) \times n_c$  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

```
254 if nargin==0
255 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 (" $\leftrightarrow$ " denotes function recursion).

1. configPatches1
2. ode15s integrator  $\leftrightarrow$  patchSys1  $\leftrightarrow$  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  $2\pi$ -periodic domain, with eight patches, spectral interpolation couples the patches, with micro-grid spacing 0.06, and with seven microscale points forming each patch.

```
275 global patches
276 patches = configPatches1(@BurgersPDE,[0 2*pi],[],8,0,0.06,7);
```

Set some initial condition, with some microscale randomness.

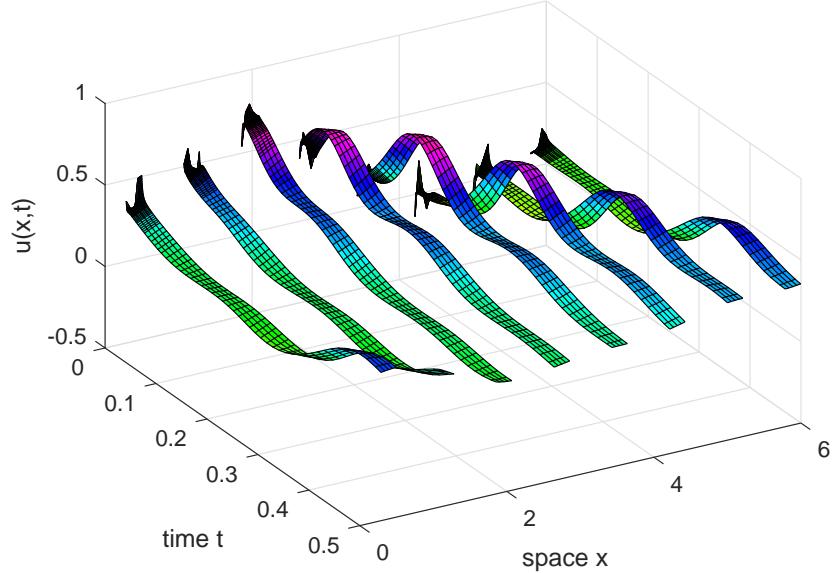
```
282 u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
```

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

```
290 if ~exist('OCTAVE_VERSION','builtin')
291 [ts,us] = ode15s(@patchSys1,[0 0.5],u0(:));
292 else % octave version
293 [ts,us] = ode0cts(@patchSys1,[0 0.5],u0(:));
294 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](#)

*Figure 3.2: field  $u(x, t)$  of the patch scheme applied to Burgers' PDE.  
Burgers PDE: patches in space, continuous time*



illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```

306 figure(1),clf
307 if 1, patches.x([1 end],:,:, :)=nan; us=us.';
308 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
309 end
310 surf(ts,patches.x(:,us)
311 view(60,40), colormap(0.8*hsv)
312 title('Burgers PDE: patches in space, continuous time')
313 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.

```

327 ifOurCf2eps(mfilename)
328 return
329 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()`).

```

15 function ut=BurgersPDE(t,u,patches)
16 u=squeeze(u); % omit singleton dimensions
17 dx=diff(patches.x(1:2)); % microscale spacing
18 i=2:size(u,1)-1; % interior points in patches
19 ut=nan+u; % preallocate output array
20 ut(i,:)=diff(u,2)/dx^2 ...

```

```

21 -30*u(i,:)*(u(i+1,:)-u(i-1,:))/(2*dx);
22 end

10 function [ts,xs] = ode0cts(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

```

### 3.1.2 Parse input arguments and defaults

```

344 p = inputParser;
345 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
346 addRequired(p,'fun',fnValidation);
347 addRequired(p,'Xlim',@isnumeric);
348 %addRequired(p,'Dom'); % nothing yet decided
349 addRequired(p,'nPatch',@isnumeric);
350 addRequired(p,'ordCC',@isnumeric);
351 addRequired(p,'dx',@isnumeric);
352 addRequired(p,'nSubP',@isnumeric);
353 addParameter(p,'nEdge',1,@isnumeric);
354 addParameter(p,'EdgyInt',false,@islogical);
355 addParameter(p,'nEnsem',1,@isnumeric);
356 addParameter(p,'hetCoeffs',[],@isnumeric);
357 addParameter(p,'parallel',false,@islogical);
358 addParameter(p,'nCore',1,@isnumeric);
359 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

365 patches.nEdge = p.Results.nEdge;
366 patches.EdgyInt = p.Results.EdgyInt;
367 patches.nEnsem = p.Results.nEnsem;
368 cs = p.Results.hetCoeffs;
369 patches.parallel = p.Results.parallel;
370 patches.nCore = p.Results.nCore;

```

Check parameters.

```

377 assert(Xlim(1)<Xlim(2) ...
378 , 'two entries of Xlim must be ordered increasing')
379 assert(patches.nEdge==1 ...
380 , 'multi-edge-value interp not yet implemented')
381 assert(2*patches.nEdge+1<=nSubP ...
382 , 'too many edge values requested')
383 if patches.nCore>1
384 warning('nCore>1 not yet tested in this version')
385 end

```

For compatibility with pre-2023 functions, if parameter Dom is `Nan`, then we set the `ratio` to be the value of the so-called `dx` parameter.

```
395 if ~isstruct(Dom), pre2023=isnan(Dom);
396 else pre2023=false; end
397 if pre2023, ratio=dx; dx=nan; end
```

Default macroscale conditions are periodic with evenly spaced patches.

```
405 if isempty(Dom), Dom=struct('type','periodic'); end
406 if (~isstruct(Dom))&isnan(Dom), Dom=struct('type','periodic'); end
```

If Dom is a string, then just set type to that string, and then get corresponding defaults for others fields.

```
414 if ischar(Dom), Dom=struct('type',Dom); end
```

Check what is and is not specified, and provide default of Dirichlet boundaries if no `bcOffset` specified when needed.

```
422 patches.periodic=false;
423 switch Dom.type
424 case 'periodic'
425 patches.periodic=true;
426 if isfield(Dom,'bcOffset')
427 warning('bcOffset not available for Dom.type = periodic'), end
428 if isfield(Dom,'X')
429 warning('X not available for Dom.type = periodic'), end
430 case {'equispace','chebyshev'}
431 if ~isfield(Dom,'bcOffset'), Dom.bcOffset=[0;0]; end
432 if length(Dom.bcOffset)==1
433 Dom.bcOffset=repmat(Dom.bcOffset,2,1); end
434 if isfield(Dom,'X')
435 warning('X not available for Dom.type = equispace or chebyshev')
436 end
437 case 'usergiven'
438 if isfield(Dom,'bcOffset')
439 warning('bcOffset not available for usergiven Dom.type'), end
440 assert(isfield(Dom,'X'),'X required for Dom.type = usergiven')
441 otherwise
442 error(['Dom.type ' is unknown Dom.type'])
443 end%switch Dom.type
```

### 3.1.3 The code to make patches and interpolation

First, store the pointer to the time derivative function in the struct.

```
455 patches.fun=fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 and  $-1$ .

```
464 assert((ordCC>=-1) & (floor(ordCC)==ordCC), ...
465 'ordCC out of allowed range integer>=-1')
```

For odd `ordCC`, interpolate based upon odd neighbouring patches as is useful for staggered grids.

```
472 patches.stag=mod(ordCC,2);
473 ordCC=ordCC+patches.stag;
474 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

```
480 if patches.stag, assert(mod(nPatch,2)==0, ...
481 'Require an even number of patches for staggered grid')
482 end
```

Third, set the centre of the patches in the macroscale grid of patches, depending upon `Dom.type`.

```
491 switch Dom.type
```

The periodic case is evenly spaced within the spatial domain. Store the size ratio in `patches`.

```
499 case 'periodic'
500 X=linspace(Xlim(1),Xlim(2),nPatch+1);
501 DX=X(2)-X(1);
502 X=X(1:nPatch)+diff(X)/2;
503 pEI=patches.EdgyInt;% abbreviation
504 if pre2023, dx = ratio*DX/(nSubP-1-pEI)*(2-pEI);
505 else ratio = dx/DX*(nSubP-1-pEI)/(2-pEI); end
506 patches.ratio=ratio;
```

In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling.<sup>2</sup>

```
514 if ordCC>0
515 [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
516 patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
517 end
```

The equi-spaced case is also evenly spaced but with the extreme edges aligned with the spatial domain boundaries, modified by the offset.

```
526 case 'equispace'
527 X=linspace(Xlim(1)+((nSubP-1)/2-Dom.bcOffset(1))*dx ...
528 ,Xlim(2)-((nSubP-1)/2-Dom.bcOffset(2))*dx ,nPatch);
529 DX=diff(X(1:2));
530 width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
531 if DX<width*0.999999
532 warning('too many equispace patches (double overlapping)')
533 end
```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors,  $X_i \propto -\cos(i\pi/N)$ , but with the

---

<sup>2</sup> **ToDo:** Might sometime extend to coupling via derivative values.

extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible ‘boundary layers’.<sup>3</sup>

```

550 case 'chebyshev'
551 halfWidth=dx*(nSubP-1)/2;
552 X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
553 X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
554 % X = (X1+X2)/2-(X2-X1)/2*cos(linspace(0,pi,nPatch));

```

Search for total width of ‘boundary layers’ so that in the interior the patches are non-overlapping Chebyshev. But the width for assessing overlap of patches is the following variable `width`.

```

563 width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
564 for b=0:2:nPatch-2
565 DXmin=(X2-X1-b*width)/2*(1-cos(pi/(nPatch-b-1)));
566 if DXmin>width, break, end
567 end
568 if DXmin<width*0.999999
569 warning('too many Chebyshev patches (mid-domain overlap)')
570 end

```

Assign the centre-patch coordinates.

```

576 X = [X1+(0:b/2-1)*width ...
577 (X1+X2)/2-(X2-X1-b*width)/2*cos(linspace(0,pi,nPatch-b)) ...
578 X2+(1-b/2:0)*width];

```

The user-given case is entirely up to a user to specify, we just force it to have the correct shape of a row.

```

587 case 'usergiven'
588 X = reshape(Dom.X,1,[]);
589 end%switch Dom.type

```

Fourth, construct the microscale grid in each patch. Reshape the grid to be 4D to suit dimensions (micro,Vars,Ens,macro).

```

599 assert(patches.EdgyInt | mod(nSubP,2)==1, ...
600 'configPatches1: nSubP must be odd')
601 i0=(nSubP+1)/2;
602 patches.x = reshape(dx*(-i0+1:i0-1)'+X ,nSubP,1,1,nPatch);

```

### 3.1.4 Set ensemble inter-patch communication

For `EdgyInt` or centre interpolation respectively,

- the right-edge/centre realisations `1:nEnsem` are to interpolate to left-edge `le`, and

---

<sup>3</sup> However, maybe overlapping patches near a boundary should be viewed as some sort of spatial analogue of the ‘christmas tree’ of projective integration and its projection to a slow manifold. Here maybe the overlapping patches allow for a ‘christmas tree’ approach to the boundary layers. Needs to be explored??

- the left-edge/centre realisations `1:nEnsem` are to interpolate to `re`.  
`re` and `li` are ‘transposes’ of each other as `re(li)=le(ri)` are both `1:nEnsem`. Alternatively, one may use the statement

```
c=hankel(c(1:nSubP-1),c([nSubP 1:nSubP-2]));
```

to correspondingly generates all phase shifted copies of microscale heterogeneity (see `homoDiffEdgy1` of [Section 3.5](#)).

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt1()`.

```
632 nE = patches.nEnsem;
633 patches.le = 1:nE;
634 patches.ri = 1:nE;
```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more then 2D, then the higher-dimensions are reshaped into the 2nd dimension.

```
646 if ~isempty(cs)
647 [mx,nc] = size(cs);
648 nx = nSubP(1);
649 cs = repmat(cs,nSubP,1);
```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```
657 if nE==1, patches.cs = cs(1:nx-1,:); else
```

But for `nEnsem > 1` an ensemble of  $m_x$  phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```
666 patches.nEnsem = mx;
667 patches.cs = nan(nx-1,nc,mx);
668 for i = 1:mx
669 is = (i:i+nx-2);
670 patches.cs(:,:,i) = cs(is,:);
671 end
672 patches.cs = reshape(patches.cs,nx-1,nc,[]);
```

Further, set a cunning left/right realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```
682 patches.le = mod((0:mx-1)' + mod(nx-2,mx),mx)+1;
683 patches.ri = mod((0:mx-1)' - mod(nx-2,mx),mx)+1;
```

Issue warning if the ensemble is likely to be affected by lack of scale separation.  
Need to justify this and the arbitrary threshold more carefully??

```

691 if ratio*patches.nEnsem>0.9, warning(...
692 'Probably poor scale separation in ensemble of coupled phase-shifts')
693 scaleSeparationParameter = ratio*patches.nEnsem
694 end

End the two if-statements.

700 end%if-else nEnsem>1
701 end%if not-empty(cs)

```

**If parallel code** then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable patches must only be referenced within an spmd-environment.*<sup>4</sup>

```

720 if patches.parallel
721 % theparpool=gcp()
722 spmd

```

Second, choose to slice parallel workers in the spatial direction.

```

729 pari = 1;
730 patches.codist=codistributor1d(3+pari);

```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```

740 switch pari
741 case 1, patches.x=codistributed(patches.x,patches.codist);
742 otherwise
743 error('should never have bad index for parallel distribution')
744 end%switch
745 end%spmd

```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

```

753 else% not parallel
754 if isfield(patches,'codist'), rmfield(patches,'codist'); end
755 end%if-parallel

```

## Fin

```

764 end% function

```

---

<sup>4</sup>If subsequently outside `spmd`, then one must use functions like `getfield(patches{1}, 'a')`.

### 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. It mostly assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables ([Section 3.1](#)) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

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

#### Input

- `u` is a vector/array of length  $n_{SubP} \cdot n_{Vars} \cdot n_{Ensem} \cdot n_{Patch}$  where there are  $n_{Vars} \cdot n_{Ensem}$  field values at each of the points in the  $n_{SubP} \times n_{Patch}$  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  $n_{SubP} \times n_{Vars} \times n_{Ensem} \times n_{Patch}$ . Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - `.x` is  $n_{SubP} \times 1 \times 1 \times n_{Patch}$  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  $n_{SubP} \cdot n_{Vars} \cdot n_{Ensem} \cdot n_{Patch}$  and the same dimensions as `u`.

Reshape the fields `u` as a 4D-array, and sets the edge values from macroscale interpolation of centre-patch values. [Section 3.3](#) describes `patchEdgeInt1()`.

```
87 sizeu = size(u);
88 u = patchEdgeInt1(u,patches);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero (as `ode15s` chokes on NaNs), then return to the user/integrator as same sized array as input.

```
98 dudt=patches.fun(t,u,patches,varargin{:});
99 dudt([1 end],:,:,:) = 0;
100 dudt=reshape(dudt,sizeu);
```

Fin.

### 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).<sup>5</sup>

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  $n_{SubP} \cdot n_{Vars} \cdot n_{Ensem} \cdot n_{Patch}$  where there are  $n_{Vars} \cdot n_{Ensem}$  field values at each of the points in the  $n_{SubP} \times n_{Patch}$  multiscale spatial grid.
- **patches** a struct largely set by configPatches1(), and which includes the following.
  - **.x** is  $n_{SubP} \times 1 \times 1 \times n_{Patch}$  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  $\geq -1$ .
  - **.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).
  - **.nEnsem** the number of realisations in the ensemble.

---

<sup>5</sup> Script patchEdgeInt1test.m verifies this code.

- `.parallel` whether serial or parallel.
- `.nCore`<sup>6</sup>

## Output

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

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```
111 if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
112 uclean=@(u) real(u);
113 else uclean=@(u) u;
114 end
```

Determine the sizes of things. Any error arising in the reshape indicates  $u$  has the wrong size.

```
122 [nx,~,~,Nx] = size(patches.x);
123 nEnsem = patches.nEnsem;
124 nVars = round(numel(u)/numel(patches.x)/nEnsem);
125 assert(numel(u) == nx*nVars*nEnsem*Nx ...
126 , 'patchEdgeInt1: input u has wrong size for parameters')
127 u = reshape(u,nx,nVars,nEnsem,Nx);
```

If the user has not defined the patch core, then we assume it to be a single point in the middle of the patch, unless we are interpolating from next-to-edge values.

These index vectors point to patches and their two immediate neighbours, for periodic domain.

```
138 I = 1:Nx; Ip = mod(I,Nx)+1; Im = mod(I-2,Nx)+1;
```

Calculate centre of each patch and the surrounding core ( $nx$  and  $nCore$  are both odd).

```
145 i0 = round((nx+1)/2);
146 c = round((patches.nCore-1)/2);
```

### 3.3.1 Periodic macroscale interpolation schemes

```
155 if patches.periodic
```

Get the size ratios of the patches, then use finite width stencils or spectral.

```
162 r = patches.ratio(1);
163 if patches.ordCC>0 % then finite-width polynomial interpolation
```

---

<sup>6</sup> ToDo: introduced sometime but not fully implemented yet, because prefer ensemble

**Lagrange interpolation gives patch-edge values** Consequently, compute centred differences of the patch core/edge averages/values for the macro-interpolation of all fields. Here the domain is macro-periodic.

```

173 if patches.EdgyInt % interpolate next-to-edge values
174 Ux = u([2 nx-1],:,:,I);
175 else % interpolate mid-patch values/sums
176 Ux = sum(u((i0-c):(i0+c),:,:,I) ,1);
177 end;

```

Just in case any last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```

185 szUx0=size(Ux);
186 szUx0=[szUx0 ones(1,4-length(szUx0)) patches.ordCC];

```

Use finite difference formulas for the interpolation, so store finite differences in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```

195 if patches.parallel
196 dmu = zeros(szUx0,patches.codist); % 5D
197 else
198 dmu = zeros(szUx0); % 5D
199 end

```

First compute differences, either  $\mu$  and  $\delta$ , or  $\mu\delta$  and  $\delta^2$  in space.

```

206 if patches.stag % use only odd numbered neighbours
207 dmu(:,:, :, I, 1) = (Ux(:,:, :, Ip)+Ux(:,:, :, Im))/2; % \mu
208 dmu(:,:, :, I, 2) = (Ux(:,:, :, Ip)-Ux(:,:, :, Im)); % \delta
209 Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
210 else % standard
211 dmu(:,:, :, I, 1) = (Ux(:,:, :, Ip)-Ux(:,:, :, Im))/2; % \mu\delta
212 dmu(:,:, :, I, 2) = (Ux(:,:, :, Ip)-2*Ux(:,:, :, I) ...
213 +Ux(:,:, :, Im)); % \delta^2
214 end%if patches.stag

```

Recursively take  $\delta^2$  of these to form successively higher order centred differences in space.

```

221 for k = 3:patches.ordCC
222 dmu(:,:, :, :, k) = dmu(:,:, :, Ip,k-2) ...
223 -2*dmu(:,:, :, I,k-2) +dmu(:,:, :, Im,k-2);
224 end

```

Interpolate macro-values to be Dirichlet edge values for each patch ([Roberts & Kevrekidis 2007](#), [Bunder et al. 2017](#)), using weights computed in `configPatches1()`. Here interpolate to specified order.

For the case where single-point values interpolate to patch-edge values: when we have an ensemble of configurations, different realisations are coupled to each other as specified by `patches.le` and `patches.ri`.

```

239 if patches.nCore==1
240 k=1+patches.EdgyInt; % use centre/core or two edges
241 u(nx,:,:,:,I) = Ux(1,:,:,:)*(1-patches.stag) ...
242 +sum(shiftdim(patches.Cwtsr,-4).*dmu(1,:,:,:,:) ,5);
243 u(1,:,:,:,I) = Ux(k,:,:,:)*(1-patches.stag) ...
244 +sum(shiftdim(patches.Cwtsl,-4).*dmu(k,:,:,:,:) ,5);

```

For a non-trivial core then more needs doing: the core (one or more) of each patch interpolates to the edge action regions. When more than one in the core, the edge is set depending upon near edge values so the average near the edge is correct.

```

254 else% patches.nCore>1
255 error('not yet considered, july--dec 2020 ??')
256 u(nx,:,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
257 + reshape(-sum(u((nx-patches.nCore+1):(nx-1),:,:,I),1) ...
258 + sum(patches.Cwtsr.*dmu),Nx,nVars);
259 u(1,:,:,:,I) = Ux(:,:,I)*(1-patches.stag) ...
260 + reshape(-sum(u(2:patches.nCore,:,:,:,I),1) ...
261 + sum(patches.Cwtsl.*dmu),Nx,nVars);
262 end%if patches.nCore

```

**Case of spectral interpolation** Assumes the domain is macro-periodic.

```
272 else% patches.ordCC<=0, spectral interpolation
```

As the macroscale fields are  $N$ -periodic, the macroscale Fourier transform writes the centre-patch values as  $U_j = \sum_k C_k e^{ik2\pi j/N}$ . Then the edge-patch values  $U_{j\pm r} = \sum_k C_k e^{ik2\pi N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$  where  $C'_k = C_k e^{ikr2\pi/N}$ . For  $\text{Nx}$  patches we resolve ‘wavenumbers’  $|k| < \text{Nx}/2$ , so set row vector  $\mathbf{ks} = k2\pi/N$  for ‘wavenumbers’  $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$  for odd  $N$ , and  $k = (0, 1, \dots, k_{\max}, (k_{\max} + 1), -k_{\max}, \dots, -1)$  for even  $N$ .

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches1()` tests that there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.<sup>7</sup>

```

296 if patches.stag % transform by doubling the number of fields
297 v = nan(size(u)); % currently to restore the shape of u
298 u = [u(:,:, :, 1:2:Nx) u(:,:, :, 2:2:Nx)];
299 stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];
300 iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
301 r = r/2; % ratio effectively halved
302 Nx = Nx/2; % halve the number of patches
303 nVars = nVars*2; % double the number of fields
304 else % the values for standard spectral
305 stagShift = 0;
306 iV = 1:nVars;
307 end%if patches.stag

```

<sup>7</sup> **ToDo:** Have not yet tested whether works for Edgy Interpolation.

Now set wavenumbers (when  $Nx$  is even then highest wavenumber is  $\pi$ ).

```

314 kMax = floor((Nx-1)/2);
315 ks = shiftdim(...
316 2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ...
317 ,-2);

```

Compute the Fourier transform across patches of the patch centre or next-to-edge values for all the fields. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le` and `patches.ri`.

```

330 if ~patches.EdgeInt
331 Cleft = fft(u(i0 ,:,:, :, :),[],4);
332 Cright = Cleft;
333 else
334 Cleft = fft(u(2 ,:,:, :, :),[],4);
335 Cright= fft(u(nx-1,:,:, :, :),[],4);
336 end

```

The inverse Fourier transform gives the edge values via a shift a fraction  $r$  to the next macroscale grid point.

```

343 u(nx,iV,patches.ri,:) = uclean(ifft(...
344 Cleft.*exp(1i*ks.*(stagShift+r)) ,[],4));
345 u(1 ,iV,patches.le,:) = uclean(ifft(...
346 Cright.*exp(1i*ks.*(stagShift-r)) ,[],4));

```

Restore staggered grid when appropriate. This dimensional shifting appears to work. Is there a better way to do this?

```

354 if patches.stag
355 nVars = nVars/2;
356 u=reshape(u,nx,nVars,2,nEnsem,Nx);
357 Nx = 2*Nx;
358 v(:,:,1:2:Nx) = u(:,:,1,:,:);
359 v(:,:,2:2:Nx) = u(:,:,2,:,:);
360 u = v;
361 end%if patches.stag
362 end%if patches.ordCC

```

### 3.3.2 Non-periodic macroscale interpolation

```

370 else% patches.periodic false
371 assert(~patches.stag, ...
372 'not yet implemented staggered grids for non-periodic')

```

Determine the order of interpolation  $p$ , and hence size of the (forward) divided difference table in  $F$ .

```

379 if patches.ordCC<1, patches.ordCC = Nx-1; end
380 p = min(patches.ordCC,Nx-1);

```

---

```
381 F = nan(patches.EdgyInt+1,nVars,nEnsem,Nx,p+1);
```

Set function values in first ‘column’ of the table for every variable and across ensemble. For `EdgyInt`, the ‘reversal’ of the next-to-edge values are because their values are to interpolate to the opposite edge of each patch.

```
391 if patches.EdgyInt % interpolate next-to-edge values
392 F(:,:, :, 1) = u([nx-1 2], :, :, I);
393 X(:,:, :, :) = patches.x([nx-1 2], :, :, I);
394 else % interpolate mid-patch values/sums
395 F(:,:, :, 1) = sum(u((i0-c):(i0+c), :, :, I) ,1);
396 X(:,:, :, :) = patches.x(i0, :, :, I);
397 end;
```

Compute table of (forward) divided differences (e.g., [Wikipedia 2022](#)) for every variable and across ensemble.

```
405 for q = 1:p
406 i = 1:Nx-q;
407 F(:,:, :, i,q+1) = (F(:,:, :, i+1,q)-F(:,:, :, i,q)) ...
408 ./ (X(:,:, :, i+q) - X(:,:, :, i));
409 end
```

Now interpolate to the edge-values at locations `Xedge`.

```
415 Xedge = patches.x([1 nx], :, :, :);
```

Code Horner’s evaluation of the interpolation polynomials. Indices `i` are those of the left end of each interpolation stencil because the table is of forward differences.<sup>8</sup> First alternative: the case of order  $p$  interpolation across the domain, asymmetric near the boundary. Use this first alternative for now.

```
431 if true
432 i = max(1,min(1:Nx,Nx-ceil(p/2))-floor(p/2));
433 Uedge = F(:,:, :, i,p+1);
434 for q = p:-1:1
435 Uedge = F(:,:, :, i,q)+(Xedge-X(:,:, :, i+q-1)).*Uedge;
436 end
```

Second alternative: lower the degree of interpolation near the boundary to maintain the band-width of the interpolation. Such symmetry might be essential for multi-D.<sup>9</sup>

```
447 else%if false
448 i = max(1,I-floor(p/2));
```

For the tapering order of interpolation, form the interior mask `Q` (logical) that signifies which interpolations are to be done at order  $q$ . This logical mask spreads by two as each order  $q$  decreases.

---

<sup>8</sup> For `EdgyInt`, perhaps interpret odd order interpolation in such a way that first-order interpolations reduces to appropriate linear interpolation so that as patches abut the scheme is ‘full-domain’. May mean left-edge and right-edge have different indices. Explore sometime??

<sup>9</sup> The aim is to preserve symmetry?? Does it?? As of Jan 2023 it only partially does—fails near boundaries, and maybe fails with uneven spacing.

---

```

457 Q = (I-1>=floor(p/2)) & (Nx-I>=p/2);
458 Imid = floor(Nx/2);

Initialise to highest divide difference, surrounded by zeros.

464 Uedge = zeros(patches.EdgyInt+1,nVars,nEnsem,Nx);
465 Uedge(:,:,:,:Q) = F(:,:,:,:,i(Q),p+1);

```

Complete Horner evaluation of the relevant polynomials.

```

471 for q = p:-1:1
472 Q = [Q(2:Imid) true(1,2) Q(Imid+1:end-1)]; % spread mask
473 Uedge(:,:,:,:Q) = F(:,:,:,:,i(Q),q) ...
474 +(Xedge(:,:,:,:Q)-X(:,:,:,:,i(Q)+q-1)).*Uedge(:,:,:,:Q);
475 end%for q
476 end%if

```

Finally, insert edge values into the array of field values, using the required ensemble shifts.

```

484 u(1 ,:,patches.le,I) = Uedge(1,:,:,:,I);
485 u(nx,:,:,Nx) = Uedge(2,:,:,:,I);

```

We want a user to set the extreme patch edge values according to the microscale boundary conditions that hold at the extremes of the domain. Consequently, may override their computed interpolation values with NaN.

```

494 u(1,:,:,:, 1) = nan;
495 u(nx,:,:,Nx) = nan;

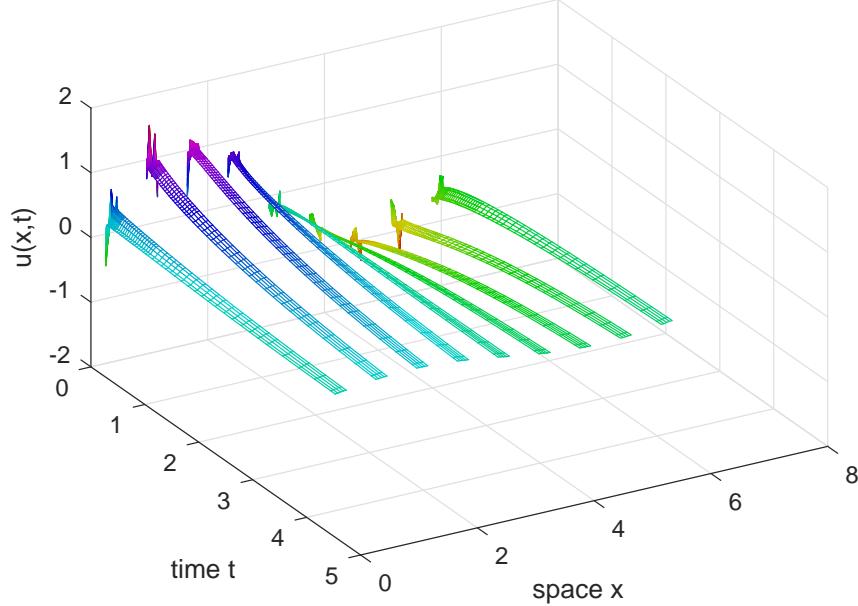
```

End of the non-periodic interpolation code.

```
501 end%if patches.periodic
```

Fin, returning the 4D array of field values.

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

|       |                                                                           |    |
|-------|---------------------------------------------------------------------------|----|
| 3.4.1 | Script to simulate via stiff or projective integration . . . . .          | 70 |
| 3.4.2 | <code>heteroDiff()</code> : heterogeneous diffusion . . . . .             | 72 |
| 3.4.3 | <code>heteroBurst()</code> : a burst of heterogeneous diffusion . . . . . | 73 |

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`  $\leftrightarrow$  `patchSys1`  $\leftrightarrow$  `heteroDiff`
3. process results

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

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

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).

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

Establish global data struct `patches` for heterogeneous diffusion on  $2\pi$ -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.

```
67 global patches
68 nPatch = 9
69 ratio = 0.2
70 nSubP = 2*mPeriod+1
71 Len = 2*pi;
72 ordCC = 4;
73 configPatches1(@heteroDiff,[0 Len],nan,nPatch ...
74 ,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.

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

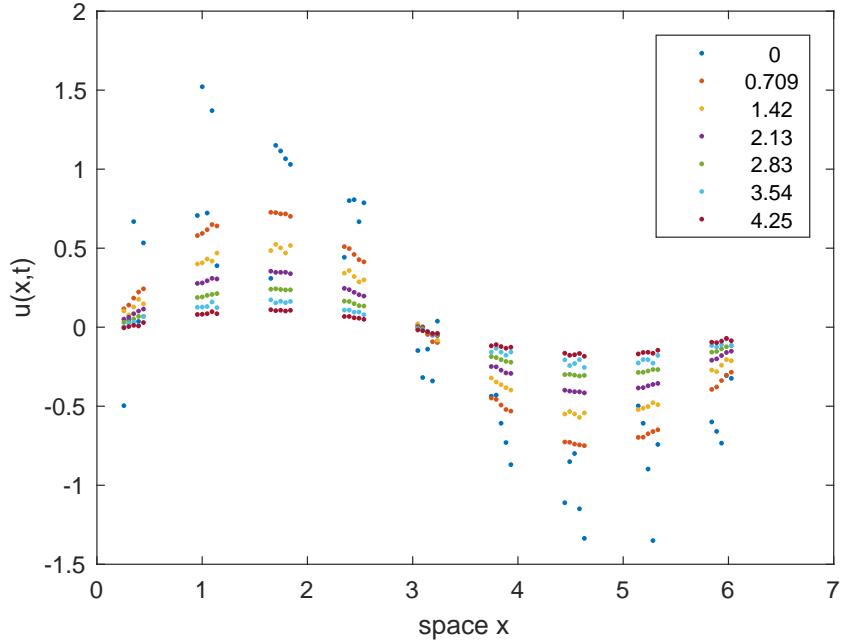
Plot the simulation in [Figure 3.3](#).

```
101 figure(1),clf
102 xs = patches.x; xs([1 end],:) = nan;
103 mesh(ts,xs(:,ucts')), view(60,40)
104 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
105 if0urCf2eps([mfilename 'CtsU'])
```

The code may invoke this integration interface.

```
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');
```

*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  $\leftrightarrow$  heteroBurst  $\leftrightarrow$  micro-integrator  $\leftrightarrow$  patchSys1  $\leftrightarrow$  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;
```

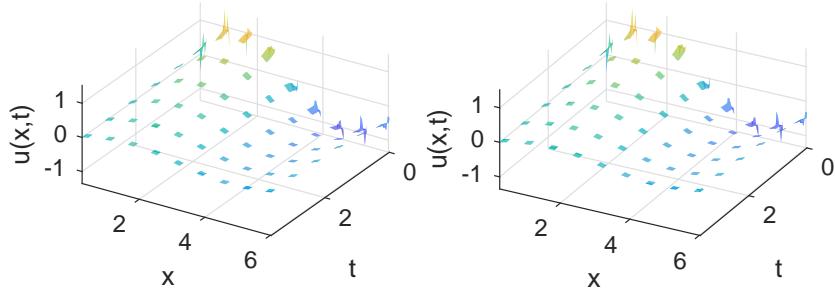
Set the desired macro- and microscale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the macroscale; the burst time is proportional to the intra-patch effective diffusion time; and lastly, the microscale time-step is proportional to the diffusion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

```

153 ts = linspace(0,2/cHomo,7)
154 bT = 3*(ratio*Len/nPatch)^2/cHomo

```

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.



```

155 addpath('..../ProjInt')
156 [us,tss,uss] = PIRK2(@heteroBurst, ts, u0(:), bT);

```

Plot the macroscale predictions to draw Figure 3.4.

```

163 figure(2),clf
164 plot(xs(:,us,'.')
165 ylabel('u(x,t)'), xlabel('space x')
166 legend(num2str(ts',3))
167 ifOurCf2eps([mfilename 'U'])

```

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

```

182 figure(3),clf
183 for k = 1:2, subplot(2,2,k)
184 surf(tss,xs(:,uss', 'EdgeColor','none')
185 ylabel('x'), xlabel('t'), zlabel('u(x,t)')
186 axis tight, view(126-4*k,45)
187 end
188 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`.

```

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

```

### 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.

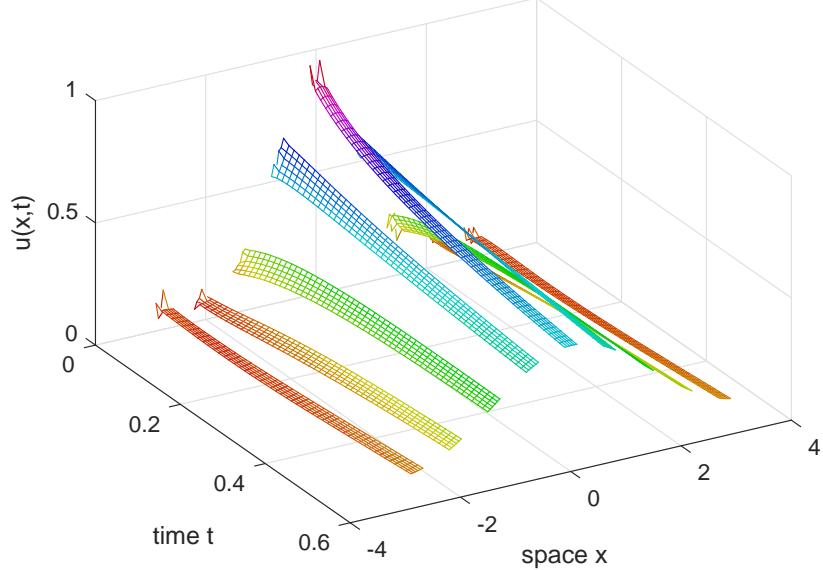
```

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

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2} \delta[c_{i+1/2} \delta u_i], \quad (3.2)$$

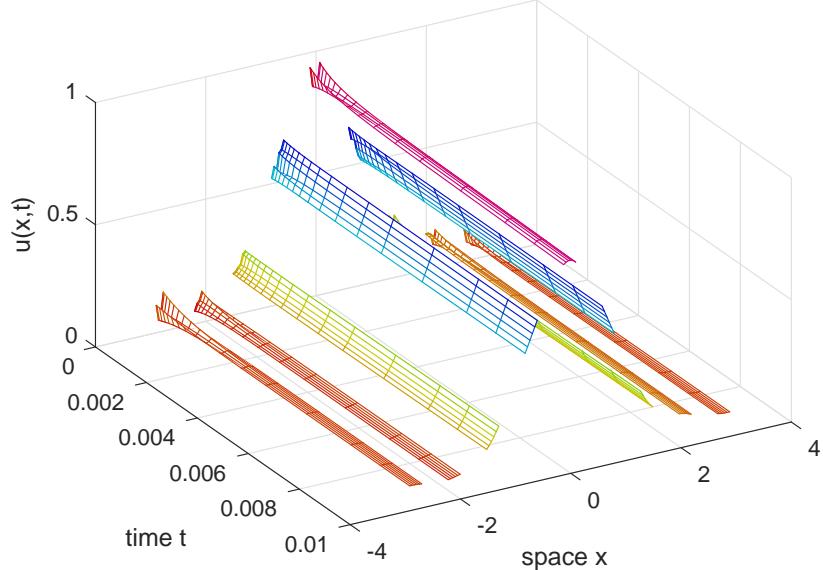
in terms of the centred difference operator  $\delta$ . The system has a microscale heterogeneity via the coefficients  $c_{i+1/2}$  which we assume to have some given known periodicity. [Figure 3.6](#) shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

#### 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  $\leftrightarrow$  patchSys1  $\leftrightarrow$  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  $2\pi$  should have near integer decay rates, the squares of  $0, 1, 2, \dots$ . Then the heterogeneity is repeated to fill each patch, and phase-shifted for an ensemble.

```

90 mPeriod = 3%randi([2 5])
91 % set random diffusion coefficients
92 cHetr=exp(0.3*randn(mPeriod,1));
93 %cHetr = [3.966;2.531;0.838;0.331;7.276];
94 cHetr = cHetr*mean(1./cHetr) % normalise

```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.2) solved on  $2\pi$ -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.EdgeInt` 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.

```

116 global patches
117 nPatch = 9
118 ratio = 0.25;

```

```

119 nSubP = mPeriod+1 %randi([mPeriod+1 2*mPeriod+2])
120 nEnsem = mPeriod % number realisations in ensemble
121 if mod(nSubP,mPeriod)==2, nEnsem=1, end
122 configPatches1(@heteroDiff, [-pi pi], nan, nPatch ...
123 , 4, ratio, nSubP, 'EdgyInt', true, 'nEnsem', nEnsem ...
124 , 'hetCoeffs', cHetr);

```

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

```

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

```

Integrate using standard integrators.

```

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

```

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

```

159 xs = squeeze(patches.x);
160 us = patchEdgeInt1(permute(reshape(us ...
161 ,length(ts),nSubP,nEnsem,nPatch) ,[2 1 3 4]));
162 usstd = squeeze(std(us,0,3));
163 us = squeeze(mean(us,3));
164 if 0, % omit interpolated edges
165 us([1 end],:,:) = nan;
166 usstd([1 end],:,:) = nan;
167 else % insert nans between patches
168 xs(end+1,:)=nan;
169 us(end+1,:,:) = nan;
170 usstd(end+1,:,:) = nan;
171 end
172 us=reshape(permute(us,[1 3 2]),[],length(ts));
173 usstd=reshape(permute(usstd,[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.

```

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

```

```

188 case 2, [~,j]=min(abs(ts(:)-linspace(ts(1),ts(end),50)));
189 end
190 figure(p),clf
191 mesh(ts(j),xs(:,),us(:,j))
192 view(60,40), colormap(0.8*hsv)
193 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
194 ifOurCf2eps([mfilename 'U' num2str(p)])
195 end
196 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.

```

209 nPatch = 13
210 ratio = 0.01;
211
212 leadingEvals=[];
213 for ord=0:2:8
214 ordInterp=ord
215 configPatches1(@heteroDiff, [-pi pi], nan, nPatch ...
216 , ord, ratio, nSubP, 'EdgyInt', true, 'nEnsem', nEnsem ...
217 , '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.

```

227 u0 = zeros(nSubP,1,nEnsem,nPatch);
228 u0([1 end],:,:, :)=nan; u0=u0(:);
229 i=find(~isnan(u0));
230 nJ=length(i);
231 Jac=nan(nJ);
232 for j=1:nJ
233 u0(i)=((1:nJ)==j);
234 dudt=patchSys1(0,u0);
235 Jac(:,j)=dudt(i);
236 end
237 nonSymmetric=norm(Jac-Jac')
238 assert(nonSymmetric<5e-9,'failed symmetry')
239 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`.

```

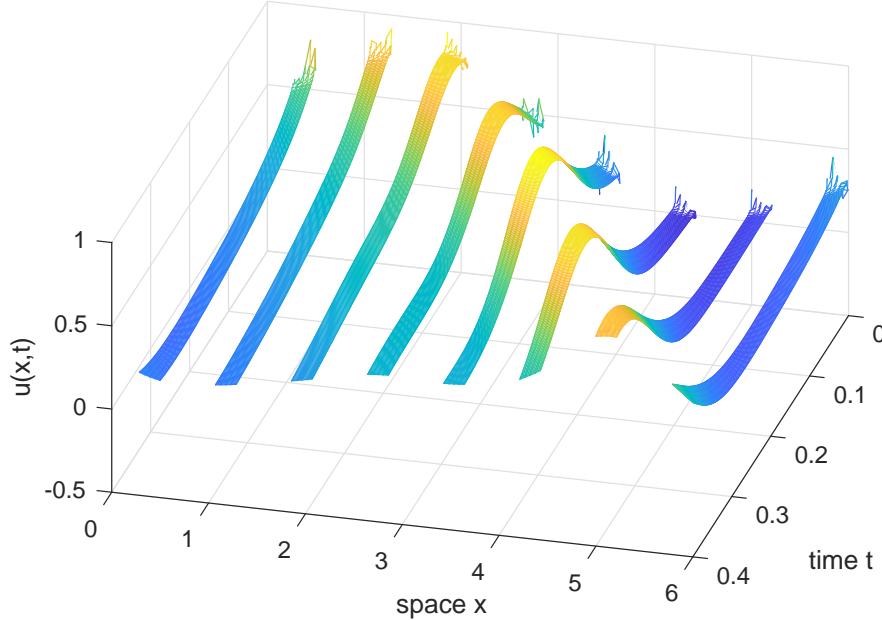
21 function ut = heteroDiff(t,u,patches)
22 dx = diff(patches.x(2:3)); % space step
23 i = 2:size(u,1)-1; % interior points in a patch

```

```
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: a short time simulation of the Burgers' map (Section 3.6.3) on patches in space. It requires many very small time-steps only just visible in this mesh.*



## 3.6 BurgersExample: simulate Burgers' PDE on patches

### Section contents

|       |                                                               |    |
|-------|---------------------------------------------------------------|----|
| 3.6.1 | Script code to simulate a microscale space-time map . . . . . | 80 |
| 3.6.2 | Alternatively use projective integration . . . . .            | 81 |
| 3.6.3 | burgersMap(): discretise the PDE microscale . . . . .         | 83 |
| 3.6.4 | burgerBurst(): code a burst of the patch map . . . . .        | 83 |

[Figure 3.2](#) shows a previous example simulation in time generated by the patch scheme applied to Burgers' PDE. The code in the example of this section similarly applies the patch scheme to a microscale space-time map ([Figure 3.8](#)), a map derived as a microscale space-time discretisation of Burgers' PDE. Then this example applies projective integration to simulate further in time.

### 3.6.1 Script code to simulate a microscale space-time map

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

1. configPatches1
2. burgerBurst  $\leftrightarrow$  patchSys1  $\leftrightarrow$  burgersMap
3. process results

Establish global data struct for the microscale Burgers' map ([Section 3.6.3](#)) solved on  $2\pi$ -periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth-order interpolation provides edge-values that couple the patches.

```

50 global patches
51 nPatch = 8
52 ratio = 0.2
53 nSubP = 7
54 interpOrd = 4
55 Len = 2*pi
56 configPatches1(@burgersMap, [0 Len], nan, nPatch, interpOrd, ratio, nSubP);

```

Set an initial condition, and simulate a burst of the microscale space-time map over a time 0.2 using the function `burgerBurst()` ([Section 3.6.4](#)).

```

64 u0 = 0.4*(1+sin(patches.x))+0.1*randn(size(patches.x));
65 [ts,us] = burgersBurst(0,u0,0.4);

```

Plot the simulation. Use only the microscale values interior to the patches by setting the edges to `nan` in order to leave gaps.

```

73 figure(1),clf
74 xs = patches.x; xs([1 end],:) = nan;
75 mesh(ts,xs(:,us'))
76 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
77 view(105,45)

```

Save the plot to file to form [Figure 3.8](#).

```
83 ifOurCf2eps([mfilename 'MapU'])
```

### 3.6.2 Alternatively use projective integration

Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of [Section 2.2](#). [Figure 3.9](#) shows the resultant macroscale prediction of the patch centre values on macroscale time-steps.

This second part of the script implements the following design.

1. `configPatches1` (done in [Section 3.6.1](#))
2. `PIRK2`  $\leftrightarrow$  `burgerBurst`  $\leftrightarrow$  `patchSys1`  $\leftrightarrow$  `burgersMap`
3. process results

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

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

Set the desired macroscale time-steps, and microscale burst length over the time domain. Then projectively integrate in time using `PIRK2()` which is second-order accurate in the macroscale time-step.

```

124 ts = linspace(0,0.5,11);
125 bT = 3*(ratio*Len/nPatch/(nSubP/2-1))^2

```

Figure 3.9: macroscale space-time field  $u(x, t)$  in a basic projective integration of the patch scheme applied to the microscale Burgers' map.

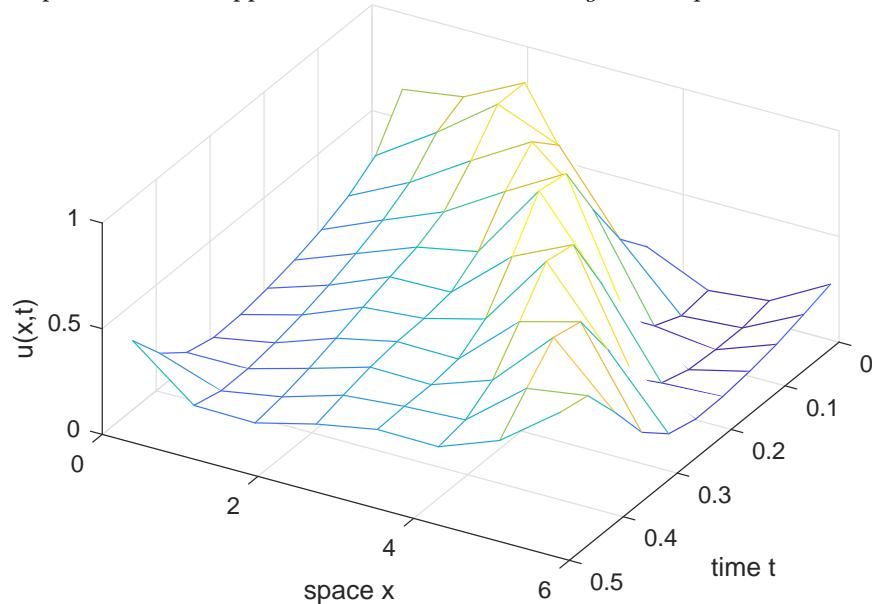
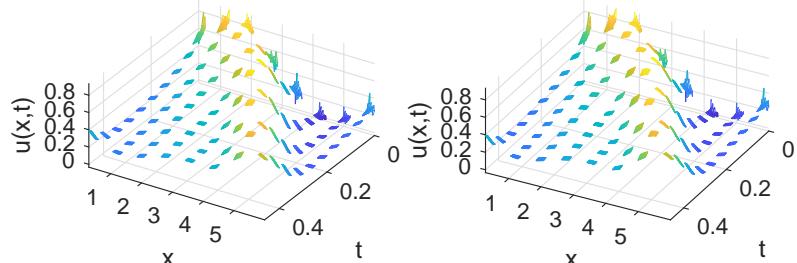


Figure 3.10: the microscale field  $u(x, t)$  during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.



```

126 addpath('..../ProjInt')
127 [us,tss,uss] = PIRK2(@burgersBurst,ts,u0(:,bT));

```

Plot and save the macroscale predictions of the mid-patch values to give the macroscale mesh-surface of Figure 3.9 that shows a progressing wave solution.

```

135 figure(2),clf
136 midP = (nSubP+1)/2;
137 mesh(ts,xs(midP,:),us(:,midP:nSubP:end)')
138 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
139 view(120,50)
140 ifUrCf2eps([mfilename 'U'])

```

Then plot and save the microscale mesh of the microscale bursts shown in Figure 3.10 (a stereo pair). The details of the fine microscale mesh are almost invisible.

```

155 figure(3),clf
156 for k = 1:2, subplot(2,2,k)

```

```

157 mesh(tss, xs(:,uss'))
158 ylabel('space x'), xlabel('time t'), zlabel('u(x,t)')
159 axis tight, view(126-4*k,50)
160 end
161 ifOutCf2eps([mfilename 'Micro'])

```

### 3.6.3 burgersMap(): discretise the PDE microscale

This function codes the microscale Euler integration map of the lattice differential equations inside the patches. Only the patch-interior values are mapped (`patchSys1()` overrides the edge-values anyway).

```

13 function u = burgersMap(t,u,patches)
14 u = squeeze(u);
15 dx = diff(patches.x(2:3));
16 dt = dx^2/2;
17 i = 2:size(u,1)-1;
18 u(i,:) = u(i,:)+dt*(diff(u,2)/dx^2 ...
19 -20*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx));
20 end

```

### 3.6.4 burgerBurst(): code a burst of the patch map

```
10 function [ts, us] = burgersBurst(ti, ui, bT)
```

First find and set the number of microscale time-steps.

```

16 global patches
17 dt = diff(patches.x(2:3))^2/2;
18 ndt = ceil(bT/dt -0.2);
19 ts = ti+(0:ndt)'*dt;

```

Use `patchSys1()` (Section 3.2) to apply the microscale map over all time-steps in the burst. The `patchSys1()` interface provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

```

29 us = nan(ndt+1,numel(ui));
30 us(1,:) = reshape(ui,1,[]);
31 for j = 1:ndt
32 ui = patchSys1(ts(j),ui);
33 us(j+1,:) = reshape(ui,1,[]);
34 end

```

Linearly interpolate (extrapolate) to get the field values at the precise final time of the burst. Then return.

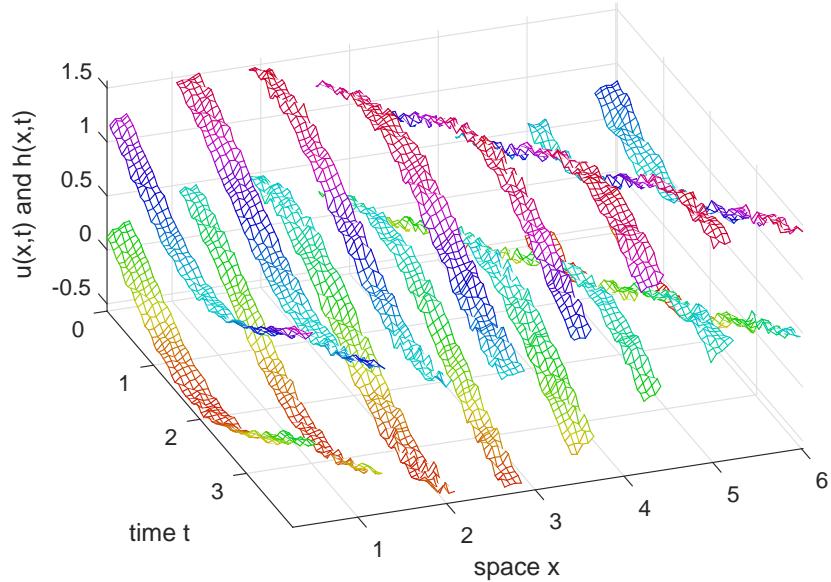
```

41 ts(ndt+1) = ti+bT;
42 us(ndt+1,:) = us(ndt,:)... + diff(ts(ndt:ndt+1))/dt*diff(us(ndt:ndt+1,:));
43
44 end

```

Fin.

*Figure 3.11: water depth  $h(x, t)$  (above) and velocity field  $u(x, t)$  (below) of the gap-tooth scheme applied to the ideal linear wave PDE (3.3) with  $f_1 = f_2 = 0$ . The microscale random component to the initial condition persists in the simulation—but the macroscale wave still propagates.*



### 3.7 waterWaveExample: simulate a water wave PDE on patches

#### Section contents

|       |                                                |    |
|-------|------------------------------------------------|----|
| 3.7.1 | Script code to simulate wave systems . . . . . | 85 |
| 3.7.2 | idealWavePDE(): ideal wave PDE . . . . .       | 87 |
| 3.7.3 | waterWavePDE(): water wave PDE . . . . .       | 88 |

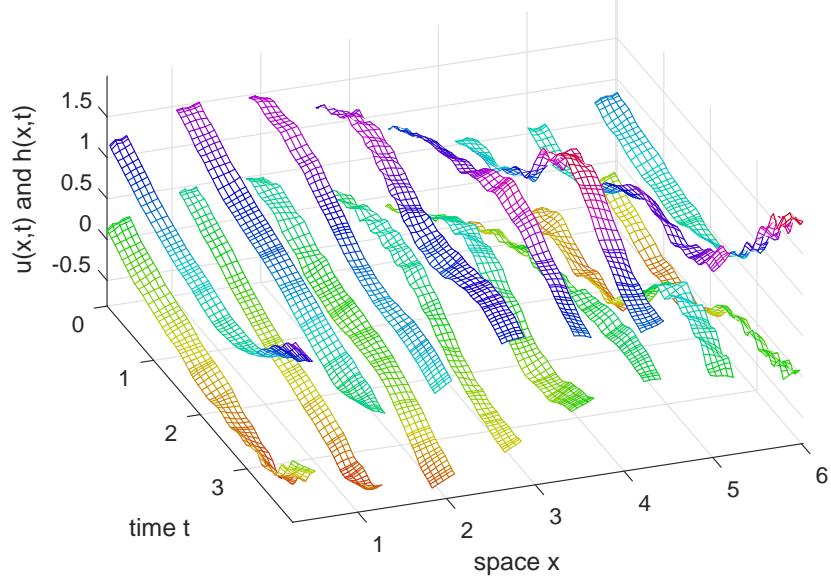
Figure 3.11 shows an example simulation in time generated by the patch scheme applied to an ideal wave PDE (Cao & Roberts 2013). The inter-patch coupling is realised by spectral interpolation of the mid-patch values to the patch edges.

This approach, based upon the differential equations coded in Section 3.7.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.7.3 that describe the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016a).

Often, wave-like systems are written in terms of two conjugate variables, for example, position and momentum density, electric and magnetic fields, and water depth  $h(x, t)$  and mean longitudinal velocity  $u(x, t)$  as herein. The approach developed in this section applies to any wave-like system in the form

$$\frac{\partial h}{\partial t} = -c_1 \frac{\partial u}{\partial x} + f_1[h, u] \quad \text{and} \quad \frac{\partial u}{\partial t} = -c_2 \frac{\partial h}{\partial x} + f_2[h, u], \quad (3.3)$$

Figure 3.12: water depth  $h(x, t)$  (above) and velocity field  $u(x, t)$  (below) of the gap-tooth scheme applied to the Smagorinski shallow water wave PDEs (3.4). The microscale random initial component decays where the water speed is non-zero due to ‘turbulent’ dissipation.



where the brackets indicate that the two nonlinear functions  $f_1$  and  $f_2$  may involve various spatial derivatives of the fields  $h(x, t)$  and  $u(x, t)$ . For example, Section 3.7.3 encodes a nonlinear Smagorinski model of turbulent shallow water (Cao & Roberts 2012, 2016a, e.g.) along an inclined flat bed: let  $x$  measure position along the bed and in terms of fluid depth  $h(x, t)$  and depth-averaged longitudinal velocity  $u(x, t)$  the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial(hu)}{\partial x}, \quad (3.4a)$$

$$\frac{\partial u}{\partial t} = 0.985 \left( \tan \theta - \frac{\partial h}{\partial x} \right) - 0.003 \frac{u|u|}{h} - 1.045u \frac{\partial u}{\partial x} + 0.26h|u| \frac{\partial^2 u}{\partial x^2}, \quad (3.4b)$$

where  $\tan \theta$  is the slope of the bed. The PDE (3.4a) represents conservation of the fluid. The momentum PDE (3.4b) represents the effects of turbulent bed drag  $u|u|/h$ , self-advection  $u\partial u/\partial x$ , nonlinear turbulent dispersion  $h|u|\partial^2 u/\partial x^2$ , and gravitational hydrostatic forcing ( $\tan \theta - \partial h/\partial x$ ). Figure 3.12 shows one simulation of this system—for the same initial condition as Figure 3.11.

For such wave-like systems, let’s implement both a staggered microscale grid and also staggered macroscale patches, as introduced by Cao & Roberts (2016b) in their Figures 3 and 4, respectively.

### 3.7.1 Script code to simulate wave systems

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

1. configPatches1, and add micro-information

- 
2. `ode15s`  $\leftrightarrow$  `patchSys1`  $\leftrightarrow$  `idealWavePDE`
  3. process results
  4. `ode15s`  $\leftrightarrow$  `patchSys1`  $\leftrightarrow$  `waterWavePDE`
  5. process results

Establish the global data struct `patches` for the PDES (3.3) (linearised) solved on  $2\pi$ -periodic domain, with eight patches, each patch of half-size ratio 0.2, with eleven micro-grid points within each patch, and spectral interpolation (-1) of ‘staggered’ macroscale patches to provide the edge-values of the inter-patch coupling conditions.

```

119 global patches
120 nPatch = 8
121 ratio = 0.2
122 nSubP = 11 %of the form 4*n-1
123 Len = 2*pi;
124 configPatches1(@idealWavePDE,[0 Len],nan,nPatch,-1,ratio,nSubP);

```

Identify which micro-grid points are  $h$  or  $u$  values on the staggered micro-grid. Also store the information in the struct `patches` for use by the time derivative function.

```

134 uPts = mod((1:nSubP)' + (1:nPatch) ,2);
135 hPts = find(uPts==0);
136 uPts = find(uPts==1);
137 patches.hPts = hPts; patches.uPts = uPts;

```

Set an initial condition of a progressive wave, and check evaluation of the time derivative. The capital letter  $U$  denotes an array of values merged from both  $u$  and  $h$  fields on the staggered grids (here with some optional microscale wave noise).

```

148 U0 = nan(nSubP,nPatch);
149 U0(hPts) = 1+0.5*sin(patches.x(hPts));
150 U0(uPts) = 0+0.5*sin(patches.x(uPts));
151 U0 = U0+0.02*randn(nSubP,nPatch);

```

**Conventional integration in time** Integrate in time using standard MATLAB/Octave stiff integrators. Here do the two cases of the ideal wave and the water wave equations in the one loop.

```

161 for k = 1:2

```

When using `ode15s/lsode` we subsample the results because micro-grid scale waves do not dissipate and so the integrator takes very small time-steps for all time.

```

169 if ~exist('OCTAVE_VERSION','builtin')
170 [ts,Ucts] = ode15s(@patchSys1,[0 4],U0(:));
171 ts = ts(1:5:end);
172 Ucts = Ucts(1:5:end,:);
173 else % octave version is slower

```

```

174 [ts,Ucts] = odeOcts(@patchSys1,[0 4],U0(:));
175 end

```

Plot the simulation.

```

181 figure(k),clf
182 xs = squeeze(patches.x); xs([1 end],:) = nan;
183 mesh(ts,xs(hPts),Ucts(:,hPts)'),hold on
184 mesh(ts,xs(uPts),Ucts(:,uPts)'),hold off
185 xlabel('time t'), ylabel('space x'), zlabel('u(x,t) and h(x,t)')
186 axis tight, view(70,45)

```

Optionally save the plot to file.

```

192 if0urCf2eps([mfilename num2str(k) 'CtsUH'])

```

For the second time through the loop, change to the Smagorinski turbulence model (3.4) of shallow water flow, keeping other parameters and the initial condition the same.

```

202 patches.fun = @waterWavePDE;
203 end

```

**Could use projective integration** As yet a simple implementation appears to fail, so it needs more exploration and thought. End of the main script.

### 3.7.2 idealWavePDE(): ideal wave PDE

This function codes the staggered lattice equation inside the patches for the ideal wave PDE system  $h_t = -u_x$  and  $u_t = -h_x$ . Here code for a staggered micro-grid, index  $i$ , of staggered macroscale patches, index  $j$ : the array

$$U_{ij} = \begin{cases} u_{ij} & i + j \text{ even}, \\ h_{ij} & i + j \text{ odd}. \end{cases}$$

The output **Ut** contains the merged time derivatives of the two staggered fields. So set the micro-grid spacing and reserve space for time derivatives.

```

24 function Ut = idealWavePDE(t,U,patches)
25 dx = diff(patches.x(2:3));
26 U = squeeze(U);
27 Ut = nan(size(U)); ht = Ut;

```

Compute the PDE derivatives only at interior micro-grid points of the patches.

```

34 i = 2:size(U,1)-1;

```

Here ‘wastefully’ compute time derivatives for both PDEs at all grid points—for simplicity—and then merge the staggered results. Since  $\dot{h}_{ij} \approx -(u_{i+1,j} - u_{i-1,j})/(2 \cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2 \cdot dx)$  as adding/subtracting one from the index of a  $h$ -value is the location of the neighbouring  $u$ -value on the staggered micro-grid.

```

46 ht(i,:) = -(U(i+1,:)-U(i-1,:))/(2*dx);

```

Since  $\dot{u}_{ij} \approx -(h_{i+1,j} - h_{i-1,j})/(2 \cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2 \cdot dx)$  as adding/subtracting one from the index of a  $u$ -value is the location of the neighbouring  $h$ -value on the staggered micro-grid.

```
56 Ut(i,:) = -(U(i+1,:)-U(i-1,:))/(2*dx);
```

Then overwrite the unwanted  $\dot{u}_{ij}$  with the corresponding wanted  $\dot{h}_{ij}$ .

```
63 Ut(patches.hPts) = ht(patches.hPts);
64 end
```

### 3.7.3 waterWavePDE(): water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (3.4). Also, regularise the absolute value appearing the the PDEs via the one-line function `rabs()`.

```
16 function Ut = waterWavePDE(t,U,patches)
17 rabs = @(u) sqrt(1e-4 + u.^2);
```

As before, set the micro-grid spacing, reserve space for time derivatives, and index the patch-interior points of the micro-grid.

```
25 dx = diff(patches.x(2:3));
26 U = squeeze(U);
27 Ut = nan(size(U)); ht = Ut;
28 i = 2:size(U,1)-1;
```

Need to estimate  $h$  at all the  $u$ -points, so into  $V$  use averages, and linear extrapolation to patch-edges.

```
36 ii = i(2:end-1);
37 V = Ut;
38 V(ii,:) = (U(ii+1,:)+U(ii-1,:))/2;
39 V(1:2,:) = 2*U(2:3,:)-V(3:4,:);
40 V(end-1:end,:) = 2*U(end-2:end-1,:)-V(end-3:end-2,:);
```

Then estimate  $\partial(hu)/\partial x$  from  $u$  and the interpolated  $h$  at the neighbouring micro-grid points.

```
47 ht(i,:) = -(U(i+1,:).*V(i+1,:)-U(i-1,:).*V(i-1,:))/(2*dx);
```

Correspondingly estimate the terms in the momentum PDE:  $u$ -values in  $U_i$  and  $V_{i\pm 1}$ ; and  $h$ -values in  $V_i$  and  $U_{i\pm 1}$ .

```
55 Ut(i,:) = -0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
56 -0.003*U(i,:).*rabs(U(i,:)./V(i,:)) ...
57 -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
58 +0.26*rabs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;
```

where the mysterious division by two in the second derivative is due to using the averaged values of  $u$  in the estimate:

$$\begin{aligned}
 u_{xx} &\approx \frac{1}{4\delta^2}(u_{i-2} - 2u_i + u_{i+2}) \\
 &= \frac{1}{4\delta^2}(u_{i-2} + u_i - 4u_i + u_i + u_{i+2}) \\
 &= \frac{1}{2\delta^2} \left( \frac{u_{i-2} + u_i}{2} - 2u_i + \frac{u_i + u_{i+2}}{2} \right) \\
 &= \frac{1}{2\delta^2} (\bar{u}_{i-1} - 2u_i + \bar{u}_{i+1}).
 \end{aligned}$$

Then overwrite the unwanted  $\dot{u}_{ij}$  with the corresponding wanted  $\dot{h}_{ij}$ .

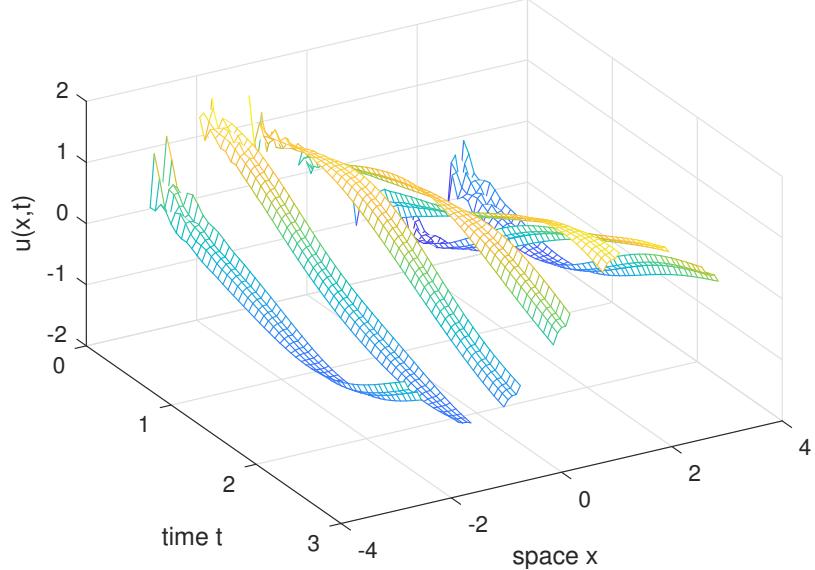
```

74 Ut(patches.hPts) = ht(patches.hPts);
75 end

```

Fin.

*Figure 3.13: wave field  $u(x, t)$  of the gap-tooth scheme applied to the weakly damped wave (3.5). The microscale random component to the initial condition persists in the simulation until the weak damping smooths the sub-patch fluctuations—but the macroscale wave still propagates.*



### 3.8 homoWaveEdgy1: computational homogenisation of a 1D wave by simulation on small patches

Figure 3.13 shows an example simulation in time generated by the patch scheme applied to macroscale wave propagation through a medium with microscale heterogeneity. The inter-patch coupling is realised by spectral interpolation of the patch’s next-to-edge values to the patch opposite edges. This coupling preserves symmetry in many systems.

Often, wave-like systems are written in terms of two conjugate variables, for example, position and momentum density, electric and magnetic fields, and water depth and mean longitudinal velocity. Here suppose the spatial microscale lattice is at points  $x_i$ , with constant spacing  $dx$ . With dependent variables  $u_i(t)$  and  $v_i(t)$ , simulate the microscale lattice, weakly damped, wave system

$$\frac{\partial u_i}{\partial t} = v_i, \quad \frac{\partial v_i}{\partial t} = \frac{1}{dx^2} \delta[c_{i-1/2} \delta u_i] + \frac{0.02}{dx^2} \delta^2 v_i, \quad (3.5)$$

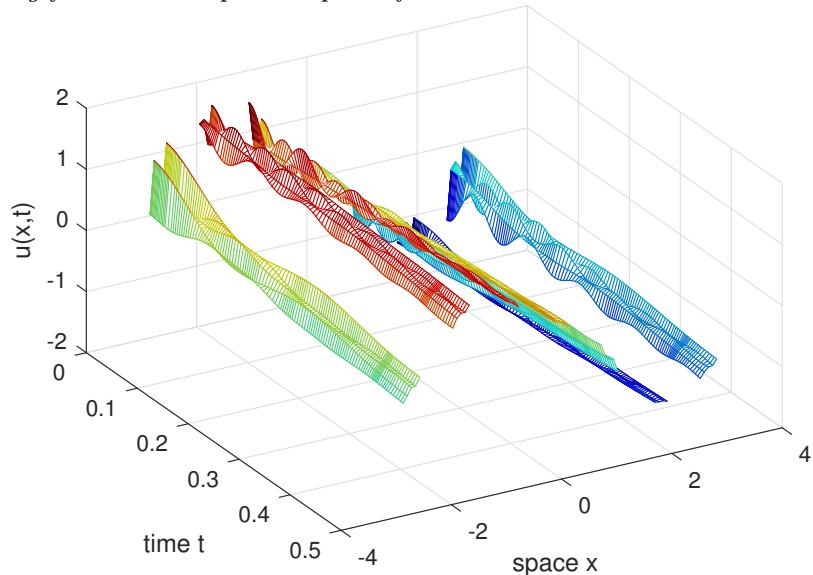
in terms of the centred difference operator  $\delta$ . The system has a microscale heterogeneity via the coefficients  $c_{i+1/2}$  which we assume to have some given known periodicity. Figure 3.13 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

#### 3.8.1 Script code to simulate heterogeneous wave systems

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

1. configPatches1, and add micro-information

Figure 3.14: wave field  $u(x, t)$  of the gap-tooth scheme applied to the weakly damped wave (3.5). Over this shorter meso-time we see the macroscale wave emerging from the damped sub-patch fast waves.



2. `ode15s`  $\leftrightarrow$  `patchSys1`  $\leftrightarrow$  `heteroWave`
3. plot the simulation
4. use `patchSys1` to check 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 waves on a domain of length  $2\pi$  have near integer frequencies, 1, 2, 3, .... Then the heterogeneity is to be repeated `nPeriodsPatch` times within each patch.

```

91 mPeriod = 3
92 cHetr = exp(1*randn(mPeriod,1));
93 cHetr = cHetr*mean(1./cHetr) % normalise
94 nPeriodsPatch=1

```

Establish the global data struct `patches` for the microscale heterogeneous lattice wave system (3.5) solved on  $2\pi$ -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 spectral interpolation (0) to provide the edge-values of the inter-patch coupling conditions. Setting `patches.EdgeyInt` to one means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```

111 global patches
112 nPatch = 7
113 ratio = 0.25
114 nSubP = nPeriodsPatch*mPeriod+2
115 configPatches1(@heteroWave, [-pi pi], nan, nPatch ...
116 , 0, ratio, nSubP, 'EdgeyInt', true, 'hetCoeffs', cHetr);

```

**Simulate** Set the initial conditions of a simulation to be that of a macroscopic progressive wave, via sin / cos, perturbed by significant random microscale noise, via randn.

```
128 uv0(:,1,1,:) = -sin(patches.x)+0.3*randn(nSubP,1,1,nPatch);
129 uv0(:,2,1,:) = +cos(patches.x)+0.3*randn(nSubP,1,1,nPatch);
```

Integrate for about half a wave period using standard stiff integrators (which do not work efficiently until after the fast waves have decayed).

```
137 if ~exist('OCTAVE_VERSION','builtin')
138 [ts,us] = ode15s(@patchSys1, [0 3], uv0(:));
139 else % octave version
140 [ts,us] = odeOcts(@patchSys1, [0 3], uv0(:));
141 end
```

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

```
153 xs = squeeze(patches.x);
154 us = patchEdgeInt1(permute(reshape(us,length(ts) ...
155 ,nSubP,2,nPatch) ,[2 3 1 4]));
156 xs(end+1,:) = nan; us(end+1,:,:,:) = nan;
157 us = reshape(permute(us,[1 4 2 3]),length(xs(:)),2,[]);
```

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 waves. The second is subsampled surface over the macroscale duration of the simulation to show the propagation of the macroscale wave over the heterogeneous lattice.

```
169 for p=1:2
170 switch p
171 case 1, j=find(ts<0.5);
172 case 2, [~,j]=min(abs(ts-linspace(ts(1),ts(end),50)));
173 end
174 figure(p),clf
175 mesh(ts(j),xs(:,1,j)), view(60,40)
176 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
177 ifOutCf2eps([mfilename 'U' num2str(p)])
178 end
```

**Compute Jacobian and its spectrum** 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 systems variables.

```
190 u0=repmat(0*patches.x,1,2); u0([1 end],:) =nan; u0=u0(:);
191 i=find(~isnan(u0));
192 nJ=length(i);
193 Jac=nan(nJ);
```

Table 3.2: example parameters and list of eigenvalues (every fourth one listed is sufficient due to symmetry): `nPatch = 7`, `ratio = 0.25`, `nSubP = 5`. The spectrum is satisfactory for weakly damped macroscale waves, and medium-damped microscale sub-patch fast waves.

```
cHetr =
 0.58459
 1.0026
 3.4253
eval =
 2.2701e-16 + 1.4225e-07i
 -0.013349 + 0.99941i
 -0.053324 + 1.9952i
 -0.11971 + 2.9838i
 -5.1527 + 19.554i
 -5.2679 + 19.695i
 -5.3383 + 19.779i
 -5.3619 + 36.632i
 -5.3722 + 36.632i
 -5.4026 + 36.631i
 -5.4514 + 36.63i
```

```
194 for j=1:nJ
195 u0(i)=((1:nJ)==j);
196 dudt=patchSys1(0,u0);
197 Jac(:,j)=dudt(i);
198 end
199 Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in [Table 3.2](#).

```
231 [evecs,evals]=eig(Jac);
232 eval=sort(diag(evals));
233 slowestEvals=eval(2:4:4*nPatch)
```

End of the main script.

### 3.8.2 `heteroWave()`: wave in heterogeneous media with weak viscous damping

This function codes the lattice heterogeneous wave equation, with weak viscosity, inside the patches. For 3D input array  $\mathbf{u}$  ( $u_{ij} = \mathbf{u}(i,1,j)$  and  $v_{ij} = \mathbf{u}(i,2,j)$ ) and 2D array  $\mathbf{x}$  (obtained in full via edge-value interpolation of `patchSys1`, [Section 3.2](#)), computes the time derivatives at each point in the interior of a patch, output in  $\mathbf{ut}$ :

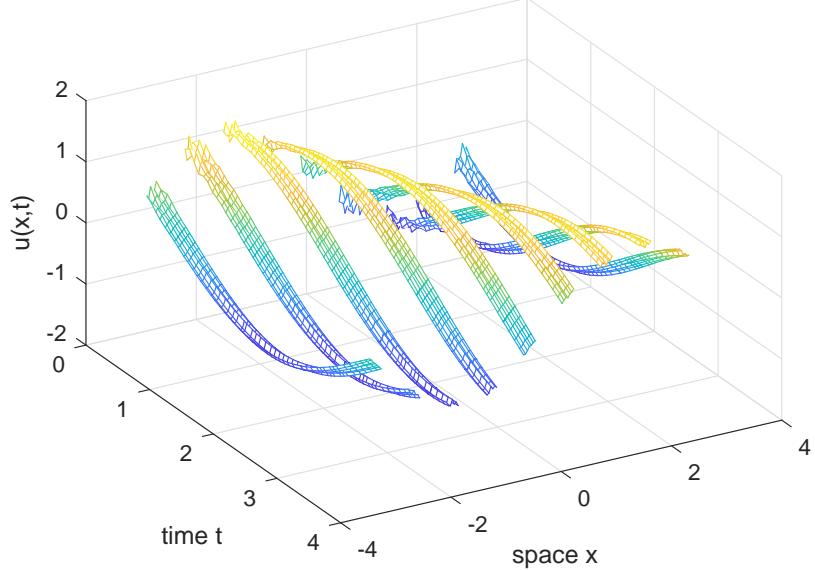
$$\frac{\partial u_{ij}}{\partial t} = v_{ij}, \quad \frac{\partial v_{ij}}{\partial t} = \frac{1}{dx^2} \delta[c_{i-1/2} \delta u_{ij}] + \frac{0.02}{dx^2} \delta^2 v_{ij}.$$

The column vector (or possibly array) of diffusion coefficients  $c_i$  have previously been stored in struct `patches`.

```
27 function ut = heteroWave(t,u,patches)
28 u = squeeze(u);
29 dx = diff(patches.x(2:3)); % space step
30 i = 2:size(u,1)-1; % interior points in a patch
31 ut = nan(size(u)); % preallocate output array
32 ut(i,1,:) = u(i,2,:); % du/dt=v then dvdt=
33 ut(i,2,:) = diff(patches.cs.*diff(u(:,1,:)))/dx^2 ...
 +0.02*diff(u(:,2,:),2)/dx^2;
34 end% function
```

Fin.

*Figure 3.15: wave field  $u(x, t)$  of the gap-tooth scheme applied to the wave (3.6). The microscale random component to the initial condition, the sub-patch fluctuations, decays, leaving the emergent macroscale wave in the heterogeneous media. This simulation uses nine patches of ‘large’ size ratio 0.25 for visibility.*



### 3.9 waveEdgy1: simulate a 1D, first-order, wave PDE on small patches

*Section contents*

|                                                                    |    |
|--------------------------------------------------------------------|----|
| 3.9.1 Script code to simulate heterogeneous wave systems . . . . . | 96 |
| 3.9.2 waveFirst(): first-order wave PDE . . . . .                  | 99 |

**Figure 3.15** 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 spectral interpolation of the patch’s next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems, and in this first-order wave PDE preserves skew-symmetry.

The first-order wave-like PDE is  $u_t = -\frac{1}{2}(cu)_x - \frac{1}{2}cu_x$ , which when  $c$  is constant becomes the canonical first-order wave PDE  $u_t = -cu_x$ . The differential operator on the right-hand side is skew-symmetric: letting  $\mathcal{D} = -\frac{1}{2}(c\cdot)_x - \frac{1}{2}c\partial_x$  then  $\int v\mathcal{D}u dx = \int -v(cu)_x - v\frac{1}{2}cu_x dx = -\int \frac{1}{2}v_x cu + \frac{1}{2}(vc)_x u dx = -\int u\mathcal{D}v dx$ .

To discretise in space, suppose the spatial microscale lattice is at points  $x_i$ , with constant spacing  $d$ . With dependent variables  $u_i(t)$ , simulate the microscale lattice, in terms of the centred difference  $\delta$  and mean  $\mu$ , wave system

$$\frac{du_i}{dt} = -\frac{1}{2d} [\delta(c_i \mu u_i) + \mu(c_i \delta u_i)] = -\frac{1}{2d} \left[ c_{i+\frac{1}{2}} u_{i+1} - c_{i-\frac{1}{2}} u_{i-1} \right]. \quad (3.6)$$

[Figure 3.15](#) shows one patch simulation of this space-time system, except it also includes a  $\nu = 0.001$  small ‘viscous’ dissipation,  $\nu\delta^2u_i/d^2$ , to weakly damp the microscale, sub-patch, fast waves.

### 3.9.1 Script code to simulate heterogeneous wave systems

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

1. configPatches1, and add micro-information
2. ode15s  $\leftrightarrow$  patchSys1  $\leftrightarrow$  waveFirst
3. plot the simulation
4. use patchSys1 to explore the Jacobian

First establish the microscale heterogeneity has (odd-valued) micro-period `mPeriod` on the lattice, and random log-normal values, normalised. This normalisation means that macroscale wave on a domain of length  $2\pi$  should have nearly integer frequencies, 0, 1, 2, . . .—except that the normalisation is exact only for periods 3 and 5. Then the heterogeneity is repeated `nPeriodsPatch` times within each patch.

```

89 mPeriod = 5 % needs to be odd for a wave
90 cHetr = exp(0.1*randn(mPeriod,1)); % 0.3 appears max reasonable
91 if mPeriod==3,
92 cHetr=cHetr*mean(cHetr.^2)/prod(cHetr) % normalise
93 elseif mPeriod==5,
94 cHetr=cHetr*mean(cHetr.^2.*cHetr([3 4 5 1 2]).^2)/prod(cHetr)
95 else cHetr=cHetr*mean(1./cHetr) % roughly normalise
96 end
97 nPeriodsPatch=1 % also needs to be odd

```

Establish the global data struct `patches` for the microscale heterogeneous lattice wave system [\(3.6\)](#) solved on  $2\pi$ -periodic domain, with nine 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 via the inter-patch coupling conditions. Setting `EdgyInt` to

- true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values); whereas
- false means the time integration appears OK, but the Jacobian is, correctly, not skew-symmetric for this case of interpolating mid-patch values.

```

120 global patches
121 nPatch = 9
122 ratio = 0.25
123 EdgyInt=true
124 nPeriodsPatch = (2-EdgyInt)*nPeriodsPatch;
125 nSubP = nPeriodsPatch*mPeriod+1+EdgyInt
126 configPatches1(@waveFirst, [-pi pi], nan, nPatch, 4 ...
 ,ratio,nSubP,'EdgyInt',EdgyInt,'hetCoeffs',cHetr);

```

Specify the weak damping of the sub-patch, fast, microscale waves.

```
135 patches.nu=0.003;
```

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

```
145 xs=squeeze(patches.x);
146 u0 = -sin(xs)+0.1*randn(nSubP,nPatch);
```

Integrate using standard stiff integrators.

```
152 if ~exist('OCTAVE_VERSION','builtin')
153 [ts,us] = ode23(@patchSys1, [0 3.5], u0(:));
154 else % octave version
155 [ts,us] = odeOcts(@patchSys1, [0 0.5], u0(:));
156 end
```

**Plot space-time surface of the simulation** Let's see the edge values of the patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate with `patchEdgeInt1` to get edge values, pad with `nans`, and reshape again.

```
167 xs(end+1,:) = nan;
168 us = patchEdgeInt1(permute(reshape(us ...
169 ,length(ts),nSubP,nPatch) ,[2 1 3]));
170 us(end+1,:,:,:)= nan;
171 us=reshape(permute(squeeze(us),[1 3 2]),[],length(ts));
```

Now plot a space-time graph. Subsample the data over the macroscale duration of the simulation to show the propagation of the macroscale wave over the heterogeneous lattice.

```
181 [~,j]=min(abs(ts-linspace(ts(1),ts(end),50)));
182 figure(1),clf
183 mesh(ts(j),xs(:,us(:,j))), view(60,40)
184 xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
185 if0urCf2eps([mfilename 'U' num2str(2)])
```

**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. Set the weak damping to zero so we explore the ideal case of the wave system (3.6).

```
198 ratio=0.01
199 nPatch=19
200 leadingFreqs=[];
201 for ord=0:2:8
202 ordInterp=ord
203 configPatches1(@waveFirst,[-pi pi],nan,nPatch,ord ...
```

Table 3.3: example parameters and list of eigenvalues (every second one listed is sufficient due to symmetry): `nPatch = 19`, `ratio = 0.03`, `nSubP = 7`. The columns are for various `ordCC`, in order: 0, spectral interpolation; 2, quadratic; 4, quartic; and 6, sixth order. Rows are ordered in the effective wavenumber of the corresponding eigenvector (the number of zero crossings).

```
cHetr =
 0.6614
 1.5758
 1.8645
 1.4600
 0.8486
leadingFreqs =
 0 0 0 0
 1.0000 0.9819 0.9996 1.0000
 2.0000 1.8574 1.9879 1.9989
 2.9999 2.5318 2.9138 2.9830
 3.9997 2.9320 3.6688 3.8910
 4.9995 3.0146 4.1015 4.5720
 5.9991 2.7705 4.0640 4.7890
 6.9985 2.2261 3.4699 4.3042
 7.9978 1.4402 2.3421 3.0200
 8.9969 0.4981 0.8278 1.0897
 698.0262 698.0852 698.0370 698.0283
 698.1548 698.1728 698.1563 698.1549
```

```
204 ,ratio,nSubP,'EdgyInt',EdgyInt,'hetCoeffs',cHetr);
205 patches.nu=0;
```

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.

```
215 u0=0*patches.x; u0([1 end],:)=nan; u0=u0(:);
216 i=find(~isnan(u0));
217 nJ=length(i);
218 Jac=nan(nJ);
219 for j=1:nJ
220 u0(i)=((1:nJ)==j);
221 dudt=patchSys1(0,u0);
222 Jac(:,j)=dudt(i);
223 end
224 nonSkewSymmetric=norm(Jac+Jac')
225 assert(nonSkewSymmetric<1e-10,'failed skew-symmetry')
226 Jac(abs(Jac)<1e-12)=0;
```

Find the eigenvalues of the Jacobian, and list for inspection in [Table 3.3](#): the spectral interpolation is effectively exact for the macroscale; quadratic interpolation is usually qualitatively good; quartic interpolation appears to

be the lowest order for quantitative accuracy.

```

268 [evecs,evals]=eig(Jac);
269 maxRealPartEvals=max(abs(real(diag(evals))))
270 assert(maxRealPartEvals<1e-10,'failed real-part zero')
271 freqs=imag(diag(evals));

```

Use a count of zero crossings in the corresponding eigenvector in order to try to sort on the spatial wavenumber.

```

279 [~,j]=sort(sum(abs(diff(sign(real(evecs))))));
280 leadingFreqs=[leadingFreqs -freqs(j(1:2:nPatch+4))];

```

End of the for-loop over orders of interpolation, and display the spectra.

```

287 end
288 disp(' spectral quadratic quartic sixth-order ...')
289 leadingFreqs = leadingFreqs

```

End of the main script.

### 3.9.2 waveFirst(): first-order wave PDE

This function codes a lattice, first-order, heterogeneous, wave PDE inside patches. Optionally adds some viscous dissipation. For 2D input arrays  $u$  and  $x$  (via edge-value interpolation of `patchSys1`, [Section 3.2](#)), computes the time derivative [\(3.6\)](#) at each point in the interior of a patch, output in  $ut$ .

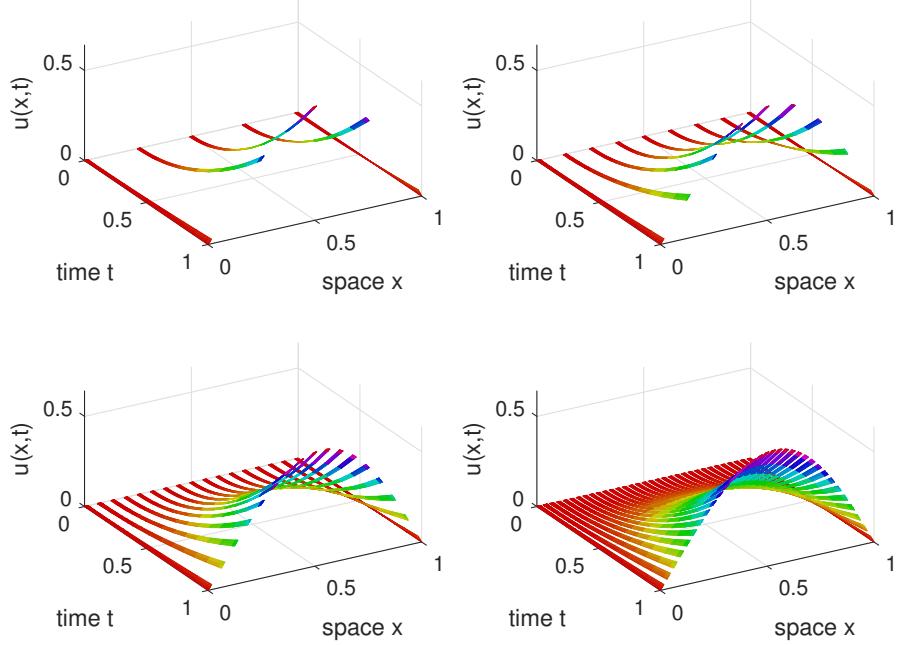
```

17 function ut = waveFirst(t,u,patches)
18 u=squeeze(u);
19 dx = diff(patches.x(2:3)); % space step
20 i = 2:size(u,1)-1; % interior points in a patch
21 ut = nan+u; % preallocate output array
22 ut(i,:) = -(patches.cs(i).*u(i+1,:)) ...
23 -patches.cs(i-1).*u(i-1,:))/(2*dx) ...
24 +patches.nu*diff(u,2)/dx^2;
25 end% function

```

Fin.

Figure 3.16: diffusion field  $u(x, t)$  of the patch scheme applied to the forced heterogeneous diffusive (3.7). Simulate for 5, 9, 17, 33 patches and compare to the full-domain simulation (65 patches, not shown).



### 3.10 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

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2} \delta[a_{i-1/2} \delta u_i] + f_i(t), \quad (3.7)$$

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

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 3.16](#). 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).

### 3.10.1 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.

```

78 clear all
79 %global OurCf2eps, OurCf2eps=true %option to save plots
80 mPeriod = 6
81 y = linspace(0,1,mPeriod+1)';
82 a = 1./(2-cos(2*pi*y(1:mPeriod)))
83 global microTimePeriod; microTimePeriod=0;

```

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

```

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

```

Loop to explore errors on various sized patches.

```

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

```

Determine indices of patches that are common in various resolutions

```

112 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 (3.7) solved on domain [0, 1], with `nP` patches, and say fourth order interpolation to provide the edge-values. Setting `patches.EdgeyInt` true means the edge-values come from interpolating the opposite next-to-edge values of the patches (not the mid-patch values).

```

127 global patches
128 ordCC = 4
129 configPatches1(@heteroDiffF,[0 1],'equispace',nP ...
130 ,ordCC,dx,nSubP,'EdgeyInt',true,'hetCoeffs',a);
131 DX = mean(diff(squeeze(patches.x(1,1,1,:))))
132 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 `nans` in between patches. For the field values (which are rows in `us`) we need to reshape, permute, interpolate to get edge values, pad with `nans`, 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

```

### 3.10.2 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,:,:,:)=0; % right-edge of rightmost is zero

```

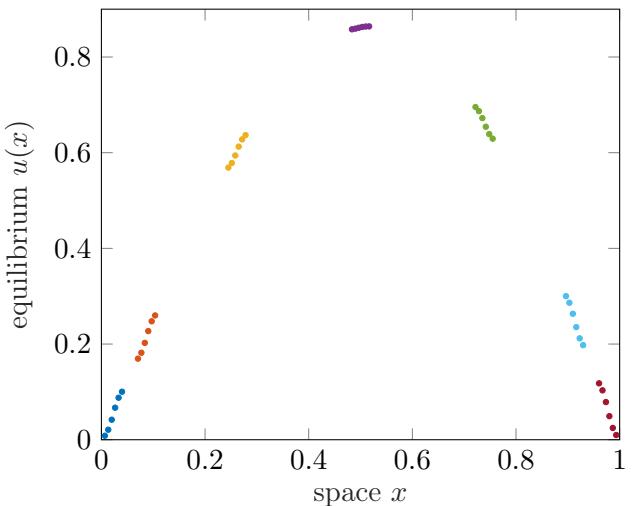
Code the microscale forced diffusion.

```

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 3.17:*  
*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.*



### 3.11 EckhardtEquilib: find an equilibrium of a 1D heterogeneous diffusion via small patches

Sections 3.10 and 3.10.2 describe details of the problem and more details of the following configuration. The aim is to find the equilibrium, Figure 3.17, 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 3.17.

**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].

```

46 clear all
47 global patches
48 %global OurCf2eps, OurCf2eps=true %option to save plots
49 mPeriod = 6
50 y = linspace(0,1,mPeriod+1)';
51 a = 1./(2-cos(2*pi*y(1:mPeriod)))
52 global microTimePeriod; microTimePeriod=0;
```

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

```

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

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.7) 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));
```

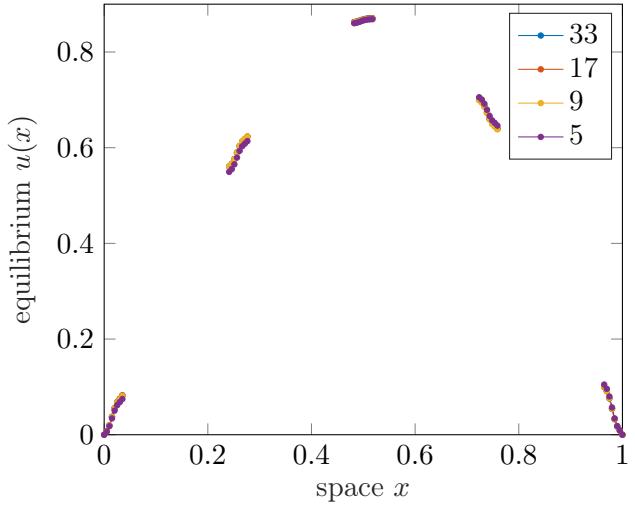
Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` ([Section 3.17](#)).

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

**Plot the equilibrium** see [Figure 3.17](#).

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

*Figure 3.18:* 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.



### 3.12 EckhardtEquilibErrs: explore errors in equilibria of a 1D heterogeneous diffusion on small patches

Section 3.11 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 3.18 and 3.19. Find mean-abs errors to be the following:

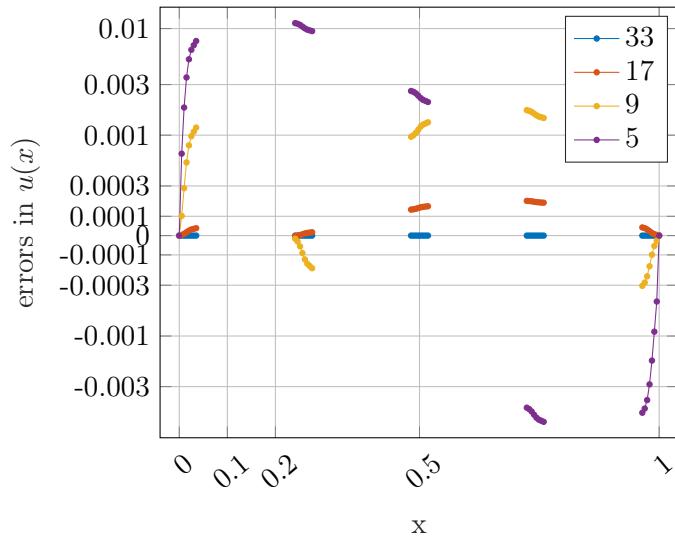
|           | $N =$        | 5    | 9    | 17   | 33   | 65    |
|-----------|--------------|------|------|------|------|-------|
| equispace | second-order | 8E-3 | 1E-2 | 1E-2 | 4E-3 | 9E-4  |
| equispace | fourth-order | 2E-3 | 7E-4 | 1E-4 | 9E-6 | 5E-7  |
| equispace | sixth-order  | 2E-3 | 2E-5 | 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’<sup>10</sup>. 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

<sup>10</sup> But the error assessment is with respect to finest patch-grid, no longer with a full domain solution

*Figure 3.19: 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).

```

136 clear all
137 global patches
138 %global OurCf2eps, OurCf2eps=true %option to save plots
139 switch 1
140 case 1, Dom.type = 'equispace'
141 case 2, Dom.type = 'chebyshev'
142 case 3, Dom.type = 'usergiven'
143 end% switch
144 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.

```

155 mPeriod = 6
156 z = (0.5:mPeriod)'/mPeriod;
157 a = 1./(2-cos(2*pi*z))
158 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  $\log 2N_{\max} = 129$  ( $\epsilon = 0.008$ ) in a few seconds:

```

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

Set the periodicity ϵ , and other microscale parameters.

177 nPeriodsPatch = 1 % any integer
178 nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
179 epsilon = 1/(nPatchMax*nPeriodsPatch+1/mPeriod)
180 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.

```

193 us=[]; xs=[]; nPs=[]
194 for log2N=log2Nmax:-1:2
195 if log2N==log2Nmax
196 Ps=linspace(1,nPatchMax ...
197 ,5-2*all(Dom.type=='chebyshev'))
198 else Ps=(Ps+1)/2
199 end

```

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

```
205 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).

```

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

```

Configure the patches:

```

226 configPatches1(@heteroDiffF,[0 1],Dom,nPatch ...
227 ,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.

```

236 if 0 %given forcing gives exact answers for ordInt=4 !
237 patches.f1 = 2*(patches.x-patches.x.^2);
238 patches.f2 = 2*0.5+0*patches.x;
239 else% simple exp-sine forcing
240 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 3.17](#)).

```

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

```

Store the solution into the `patches`, and give magnitudes— $\text{Inf}$  norm is  $\max(\text{abs}())$ .

```

279 normSoln = norm(uSoln,Inf)
280 normResidual = norm(theRes(uSoln),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 = [nP;nP];

```

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.

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

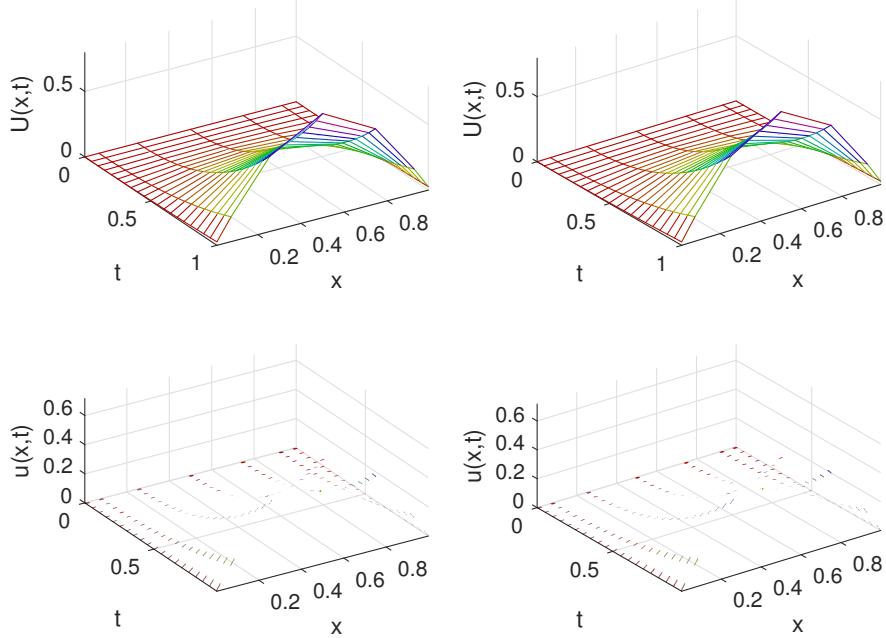
 Reshape solution field.

322 figure(1),clf
323 plot(x(:,u,'.-'), legend(num2str(nPs))
324 xlabel('space x'), ylabel('equilibrium $u(x)$')
325 ifOuRcf2tex([mfilename 'us'])%optionally save
```

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

```
333 err = u(:,1)-u;
334 figure(2), clf
335 h=plot(x(:,err,'.-'); legend(num2str(nPs))
336 quasiLogAxes(h,10,sqrt(prod(meanAbsErrs(2:3))))
337 xlabel('space x'), ylabel('errors in $u(x)$')
338 ifOuRcf2tex(mfilename)%optionally save
```

*Figure 3.20: diffusion field  $u(x, t)$  of the patch scheme applied to the forced space-time heterogeneous diffusive (3.8). 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.*



### 3.13 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

$$\frac{\partial u_i}{\partial t} = \frac{1}{dx^2} \delta[a_{i+1/2}(t) \delta u_i] + f_i(t), \quad (3.8)$$

in terms of the centred difference operator  $\delta$ . The system has a microscale heterogeneity via the coefficients  $a_{i+1/2}$  which has given periodicity  $\epsilon$  in space, and periodicity  $\epsilon^2$  in time. [Figure 3.20](#) 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’.

### 3.13.1 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.

```

98 clear all
99 %global OurCf2eps, OurCf2eps=true %option to save plots
100 mPeriod = 6
101 y = linspace(0,1,mPeriod+1)';
102 a = (3+cos(2*pi*y(1:mPeriod)))/30
103 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.

```

116 nPeriodsPatch = 2 % any integer
117 rEpsilon = 100
118 dx = 1/(mPeriod*rEpsilon+1)
119 nSubP = nPeriodsPatch*mPeriod+2
120 tol=1e-9;

```

Set the time periodicity (global).

```

126 global microTimePeriod
127 microTimePeriod = 1/rEpsilon^2

```

Establish the global data struct `patches` for the microscale heterogeneous lattice diffusion system (3.8) 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).

```

144 nPatch = 7
145 ordCC = 4

```

```

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

```

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

```

158 if 0 % given forcing
159 patches.f1=2*(patches.x-patches.x.^2);
160 patches.f2=2*0.5+0*patches.x;
161 else% simple sine forcing
162 patches.f1=sin(pi*patches.x);
163 patches.f2=pi/2*sin(pi*patches.x);
164 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.

```

175 u0 = 0*patches.x;
176 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  $\beta$  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.

```

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

```

Time the projective integration simulation.

```

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

```

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

```

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

```

```

219 figure(1),clf
220 for k = 1:2, subplot(2,2,k)
221 mesh(ts,Xs(:,Us'))
222 ylabel('x'), xlabel('t'), zlabel('U(x,t)')
223 colormap(0.8*hsv), axis tight, view(62-4*k,45)
224 end

```

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

```

232 xs([1 end],:) = nan;
233 for k = 1:2, subplot(2,2,2+k)
234 surf(tss,xs(:,uss', 'EdgeColor','none')
235 ylabel('x'), xlabel('t'), zlabel('u(x,t)')
236 colormap(0.7*hsv), axis tight, view(62-4*k,45)
237 end
238 ifOrCf2eps(mfilename)

```

### 3.13.2 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).

```

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

```

### 3.14 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  $\alpha$ , we explore the dynamics of  $\vec{M}(x, t)$  governed by the nonlinear Landau–Lifshitz PDE (Leitenmaier & Runborg 2021, (1.1)) <sup>11</sup>

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

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 3.21](#) 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

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

in terms of the centred difference operator  $\delta$ . 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 3.21](#) 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  $\alpha$  should be small. If the initial conditions are smooth, then `ode15s` has no problems for  $\alpha = 0.001$ . <sup>12</sup>

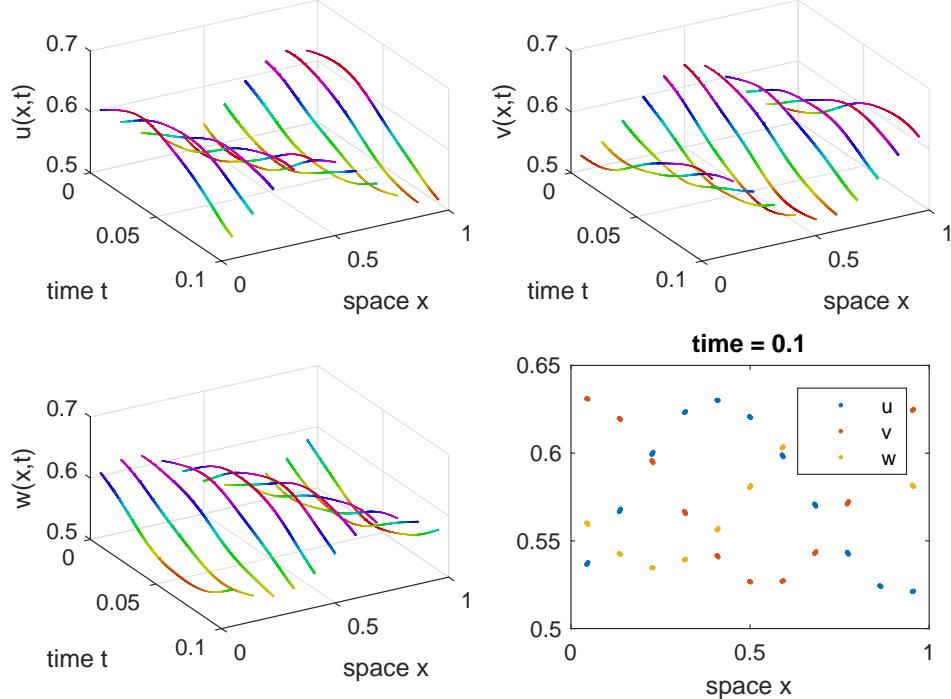
```
89 global alpha ex5p1
90 ex5p1 = 0; % set to 1 for L&O example of p.27
91 alpha = 0.001 % phenomenological damping parameter
```

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

<sup>11</sup> Recall  $a \times (b \times c) = (a \cdot c)b - (a \cdot b)c$

<sup>12</sup> 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 3.21: 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 \(2021\)](#).



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

### 3.14.1 Script code to simulate heterogeneous diffusion systems

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

1. configPatches1
2. ode15s  $\leftrightarrow$  patchSys1  $\leftrightarrow$  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.

```
125 dx = 1/2000 %1/6000 %p.27
126 mPeriod = round(epsilon/dx)
127 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).

```

143 global patches
144 nPatch = 11 %24 %p.6, odd is slightly cleaner
145 nSubP = mPeriod+2
146 ratio = nPatch*epsilon
147 configPatches1(@heteroLanLif1D,[0 1],nan,nPatch ...
148 ,0,ratio,nSubP,'EdgyInt',true ...
149 ,'hetCoeffs',a);
150 assert(abs(dx-diff(patches.x(2:3)))<1e-10 ...
151 , '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  $|\vec{M}(x, 0)| = 1$ .

```

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

Integrate using standard integrators.

```

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

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

```

185 xs = squeeze(patches.x);
186 Ms = reshape(Ms,length(ts),nSubP,3,nPatch);
187 Ms(:,[1 end],:,:) = nan; % nan patch edges
188 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  $10^{-5}$ .

```

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

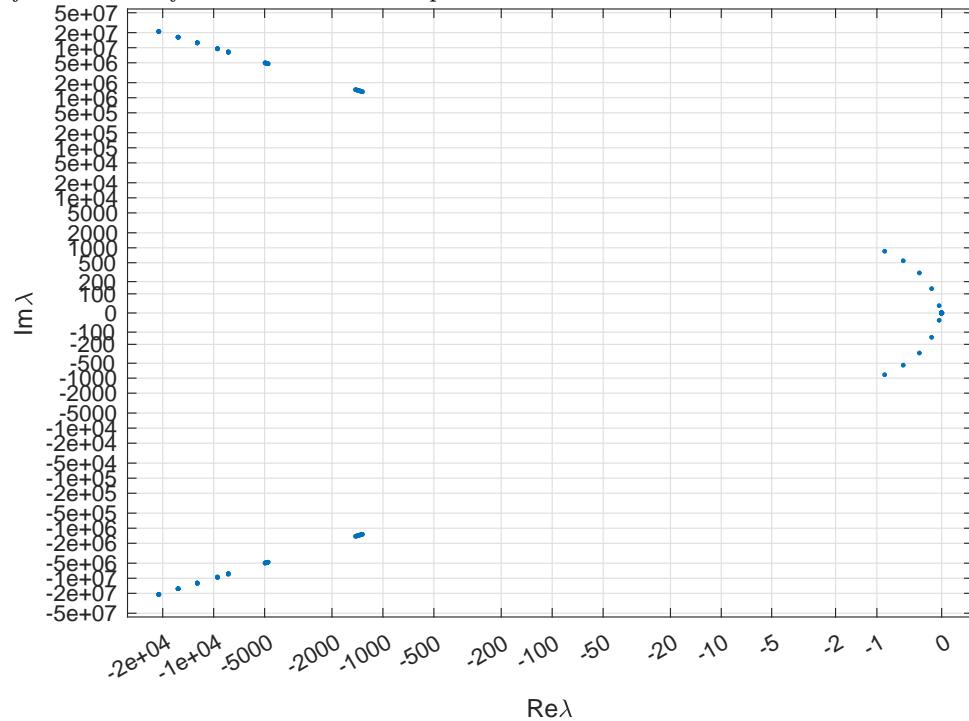
**Plot space-time surface of the simulation** Choose whether to save some plots, or not.

```

208 global OurCf2eps
209 OurCf2eps = false;
```

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

Figure 3.22: 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.



```

217 figure(1),clf
218 if length(ts)>50
219 [~,j]=min(abs(ts(:)-linspace(ts(1),ts(end),50)));
220 else j=1:length(ts); end
221 uvw='uvw';
222 for p=1:3
223 subplot(2,2,p)
224 mesh(ts(j),xs(:,Ms(:,j,p)))
225 view(60,40), colormap(0.8* hsv)
226 xlabel('time t'), ylabel('space x')
227 zlabel(['uvw(p) '(x,t)''])
228 end

```

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

```

235 subplot(2,2,4)
236 plot(xs(:,squeeze(Ms(:,end,:))),'.')
237 xlabel('space x'), legend(uvw(1),uvw(2),uvw(3))
238 title(['time = ' num2str(ts(end),4)])
239 if0urCf2eps([mfilename 'uvw'])

```

### 3.14.2 Spectrum of the coded patch system

It appears the spectrum has the following properties as shown by [Figure 3.22](#), with  $N = \text{nPatch}$  and  $n = \text{nSubP} - 2$ , 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  $\epsilon$ —the periodicity and patch size.

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

```

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

```

Form the Jacobian by numerical differentiation.

```

288 delta=1e-7;
289 nJac=length(i);
290 Jac=nan(nJac);
291 for j=1:nJac
292 M=Me; M(i(j))=M(i(j))+delta;
293 fj=patchSys1(0,M(:));
294 Jac(:,j)=(fj(i)-f0(i))/delta;
295 end

```

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

```

302 eval = eig(Jac);
303 [~,k] = sort(abs(eval));
304 eval = eval(k);
305 nZero = sum(abs(eval)<1)
306 nCent = sum(abs(real(eval))<1e5*alpha)
307 nSlow = sum(abs(eval)<1e5)

```

Plot the spectrum of eigenvalues on quasi-log axes.

```

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

```

### 3.14.3 `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)):

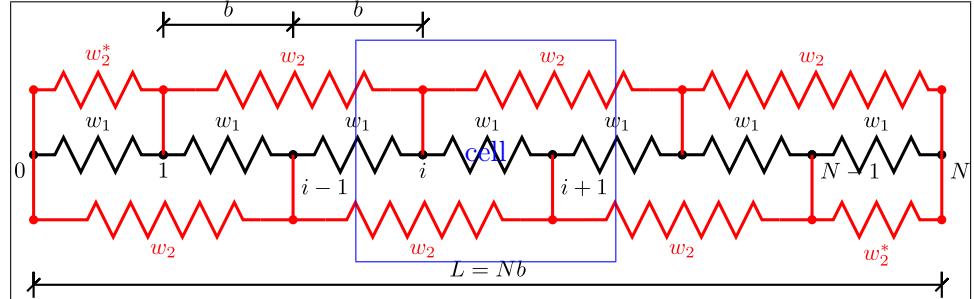
```

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 3.23: 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  $i$ .



### 3.15 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 3.23 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 3.23) and for every  $i$ ,

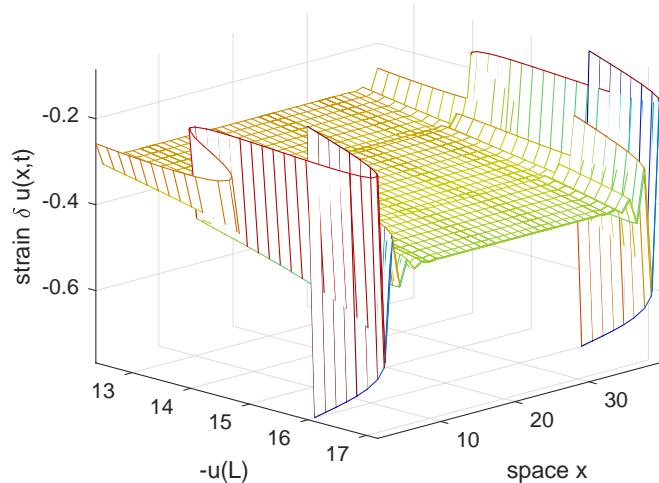
$$\begin{aligned} \epsilon_i^p &:= \frac{1}{pb}(u_{i+p/2} - u_{i-p/2}), & \sigma_i^p &:= w_p'(\epsilon_i^p), \\ \frac{\partial^2 u_i}{\partial t^2} &= \sum_{p=1}^2 \frac{1}{pb}(\sigma_{i+p/2}^p - \sigma_{i-p/2}^p), & w_p'(\epsilon) &:= \epsilon - M_p \epsilon^3 + \epsilon^5. \end{aligned} \quad (3.9)$$

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 3.24 and 3.27(b)); and
- large scale ‘instability’ (structure) arises with  $M_1 := -1$  and  $M_2 := 3$  (Figures 3.26 and 3.27(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 3.24 shows boundary layers with microscale variations develop for  $u(L) < -13$ . This figure plots a strain  $\epsilon$  as the strain is nearly constant across the interior, so the boundary layers show up clearly. As  $u(L)$  decreases further, Figure 3.24 shows the family of equilibria form complicated folds. Table 3.4 lists that MatCont also reports some branch

*Figure 3.24:* 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 3.27\(b\)](#)).



[Table 3.4:](#) Interesting equilibria for the cases of small scale instability:  $M_1 := 2$ ,  $M_2 := 1$  ([Figures 3.24](#) and [3.27\(b\)](#)). The rightmost column gives the  $-u(L)$  parameter values for corresponding critical points in the three-patch code ([Figure 3.25](#)).

| $-u(L)$ | 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 |

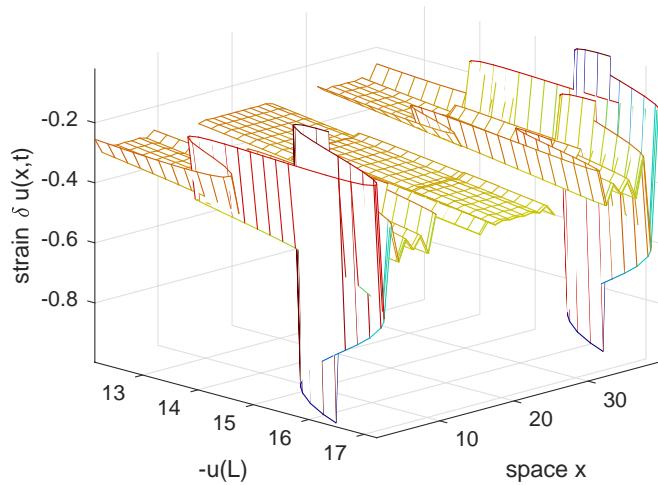
points and neutral saddle equilibria in this same regime (see [Figure 3.27\(b\)](#)). I have not yet followed any of the branches.

The previous paragraph’s discussion is for a full domain simulation, albeit done through an imposed computational framework of physically abutting patches. [Figure 3.25](#) 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 3.25](#) shows the patch scheme reasonably resolves these. [Table 3.4](#) 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.

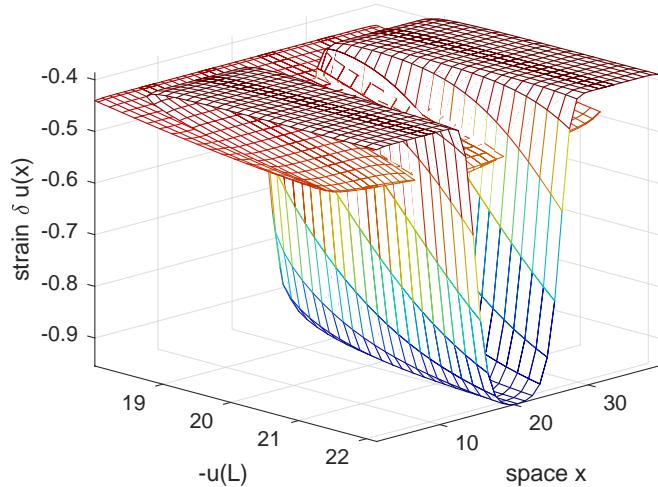
Importantly, MatCont is about *ten times quicker to execute on the patches* than on the full domain code. This speed-up indicates that on larger scale problems the patch scheme could be very useful in continuation explorations.

**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 3.26](#) shows an interior region of higher magnitude strain develops. Again, this figure plots a strain  $\epsilon$  as the strain is nearly constant across the domain, so the interior structure shows up clearly. As  $u(L)$

*Figure 3.25:* 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 3.24](#). But it is computed ten times quicker.



*Figure 3.26:* 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 3.27\(a\)](#).



decreases further, [Figure 3.26](#) shows the family of equilibria form complicated folds. [Table 3.5](#) lists that MatCont also reports some branch points and neutral saddle equilibria in this regime (see [Figure 3.27\(a\)](#)). 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.

### 3.15.1 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,

|  | $-u(L)$ | 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 physical displacements form the array

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

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

```

229 clear all
230 global b M vis
231 b = 1 % separation of lattice points
232 N = 42 % # lattice steps in L
233 L = b*N % length of domain

```

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

```

240 theCase = 2
241 switch theCase
242 case 1, M = [0 0 0 0] % linear spring coefficients
243 case 2, M = [2 1 1 1] % micro scale instability??
244 case 3, M = [-1 3 1 1] % large scale instability??
245 end% switch
246 vis = 0.1 % does not appear to affect the equilibria
247 tEnd = 25

```

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

```

253 edgyInt = true
254 nSubP = 6, nPatch = 5 % gives full-domain on N=42, dx=2
255 %nSubP = 6, nPatch = 3 % patches for some crude comparison

```

Establish the global data struct `patches` for the microscale heterogeneous lattice elasticity system (3.9). Solved with `nPatch` patches, and interpolation (as high-order as possible) to provide the edge-values of the inter-patch coupling conditions.

```

268 global patches
269 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

```

### 3.15.2 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 `nans` 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

```

### 3.15.3 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)

```

```
413 iP=1:dP:nPars;
414 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.

```
422 uvs=nan(numel(iP),numel(u0));
423 uvs(:,patches.i)=uv(1:nVars,iPars(iP))';
424 uvs = reshape(permute(reshape(uvs ...
425 ,length(muLP),nSubP,4,1,nPatch) ,[2 3 1 4 5]) ,nSubP,[],1,nPatch);
426 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.

```
434 us = reshape(permute(uvs(:,1:2,:,:,:) ...
435 ,[2 1 4 3]) ,2*nSubP,nPatch,[]);
436 us(end+1,:,:)= nan;
437 us = reshape(us,[],length(muLP));
```

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

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

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

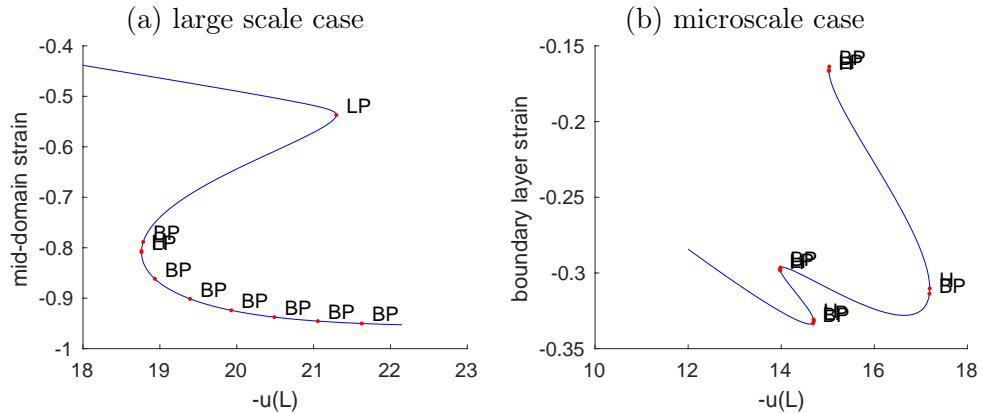
```
454 figure(5), clf()
455 mesh(muLP, xs(1:end-1),diff(us))
456 view(45,20), colormap(0.8*jet), axis tight
457 xlabel('u(L)'), ylabel('space x'), zlabel('strain \delta u(x)')
458 ifOurCf2eps(['Comb22diffu' num2str(theCase)], [12 9])%optionally save
```

**Labelled parameter plot** Get the labelled 2D plots of [Figure 3.27](#) 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??).

```
488 uvs = nan(numel(muLs),numel(u0));
489 uvs(:,patches.i) = uv(1:nVars,:)';
490 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 3.27: cross-sections through Figures 3.24 and 3.26: (a) large scale case, at the mid-point in space of Figure 3.26; (b) microscale case, in a boundary layer of Figure 3.24. These cross-sections are labelled with the various critical points.*



```

499 if theCase==2, thePatch=1;
500 else thePatch=(nPatches+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 if0urCf2eps(['Comb22cpl' num2str(theCase)], [9 7])%optionally save

```

### 3.15.4 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

```

### 3.15.5 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
```

### 3.15.6 `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  $j$ th point in the  $i$ th 2-cell in the  $I$ th 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,:,:,:)=0;
596 v(1,:,:,:)=0;
597 u(end,:,:,:)=muL;
598 v(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
```

### 3.16 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
```

### 3.17 quasiLogAxes(): transforms plot to quasi-log axes

This function rescales and labels the axes of the given 2D 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.

Herein  $x, y$  denotes the original data scale, and  $h, v$  denotes nonlinearly transformed quantities.

```
20 function quasiLogAxes(handle,xScale,yScale)
```

#### Input

- **handle**: handle to your plot to transform, for example, obtained by `handle=plot(...)`
- **xScale** (optional, default 1): let  $x$  denote every horizontal coordinate, then transform the plot-data with the `asinh()` function so that
  - for  $|x| \lesssim x_{\text{scale}}$  the horizontal axis scaling is approximately linear, whereas
  - for  $|x| \gtrsim x_{\text{scale}}$  the horizontal axis scaling is approximately signed-logarithmic.
- **yScale** (optional, default 1): corresponds to **xScale** for the vertical axis scaling.
- axis limits (optional): if the axis limits of the plot do not 'fit' the plot data, then we assume you have set the axis limits, in which case these limits are used (horizontal and vertical are considered separately).

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

```
48 if nargin==0
49 % generate some data
50 n=99; fast=(rand(n,1)<0.8);
51 z = -rand(n,1).*(1+1e3*fast)+1i*randn(n,1).*(5+1e2*fast);
52 % plot data and transform axes
53 handle = plot(real(z),imag(z),'o');
54 xlabel('real-part'), ylabel('imag-part')
55 quasiLogAxes(handle,1,10);
56 return
57 end% example
```

Default values for scaling.

```
65 if nargin<3, yScale=1; end
66 if nargin<2, xScale=1; end
```

**Output** None, just the transformed plot.

Get current limits of the plot so we can attempt to detect if a user has set some limits that we should keep.

```
77 lims0=axis;
```

Find overall factors so the data is nonlinearly mapped to order oneish—so that then pgfplots et al. do not think there is an overall scaling factor on the axes.

```
86 xFac=0; yFac=0;
87 for k=1:length(handle)
88 temp = asinh(handle(k).XData/xScale);
89 xFac = max(xFac, max(abs(temp(:)),[],'omitnan'));
90 temp = asinh(handle(k).YData/yScale);
91 yFac = max(yFac, max(abs(temp(:)),[],'omitnan'));
92 end%for
93 xFac=9/xFac; yFac=9/yFac;
```

Scale the plot data in the 2D plot handle. Give an error if it appears that the plot-data has already been transformed.

```
102 for k=1:length(handle)
103 assert(~strcmp(handle(k).UserData,'quasiLogAxes'), ...
104 'Replot graph, as it appears plot data is already transformed')
105 handle(k).XData = xFac*asinh(handle(k).XData/xScale);
106 handle(k).YData = yFac*asinh(handle(k).YData/yScale);
107 handle(k).UserData = 'quasiLogAxes';
108 end%for
109 lims0=[xFac*asinh(lims0(1:2)/xScale) yFac*asinh(lims0(3:4)/yScale)];
```

Get limits of nonlinearly transformed data, and reset with 4% padding around all margins—crude but serviceable.

```
116 axis tight; lims=axis;
117 dl=0.04*diff(lims); %dl(abs(dl)<4e-5) = 1;
118 lims = lims+[-dl(1) +dl(1) -dl(3) +dl(3)];
```

But if the range is too different from the original, then restore the original. Then set the limits.

```
125 dl=diff(lims); d10=diff(lims0);
126 for k=[1 3], if dl(k)<0.5*d10(k) | dl(k)>2*d10(k),
127 lims(k:k+1)=lims0(k:k+1);
128 end, end
129 axis(lims)
```

**Horizontal scaling** Get the order of magnitude of the horizontal data.

```
137 hmax=max(abs(lims(1:2)));
138 hmag=floor(log10(xScale*sinh(hmax/xFac)));
```

Form a range of ticks, geometrically spaced, trim off the small values that would be too dense near zero (omit those within 6% of `hmax`).

```

146 ticks=10.^ (hmag+(-7:0));
147 j=find(ticks>xScale*sinh(0.06*hmax/xFac));
148 nj=length(j);
149 if nj<3, ticks=[1;2;5]*ticks(j);
150 elseif nj<5, ticks=[1;3]*ticks(j);
151 else ticks=ticks(j);
152 end
153 ticks=sort([0;ticks(:);-ticks(:)]);

```

Set the ticks in place according to the transformation. Getting the ‘parent’ gives the axes of the plot (`gca`).

```

160 theaxes = get(handle(1), 'parent');
161 set(theaxes, 'Xtick', xFac*asinh(ticks/xScale) ...
162 , 'XtickLabel', cellstr(num2str(ticks,4)) ...
163 , 'XTickLabelRotation', 40)

```

**Vertical scaling** Do the same for the vertical axis.

```

171 vmax=max(abs(lims(3:4)));
172 vmag=floor(log10(yScale*sinh(vmax/yFac)));
173 ticks=10.^ (vmag+(-7:0));
174 j=find(ticks>yScale*sinh(0.06*vmax/yFac));
175 nj=length(j);
176 if nj<3, ticks=[1;2;5]*ticks(j);
177 elseif nj<5, ticks=[1;3]*ticks(j);
178 else ticks=ticks(j);
179 end
180 ticks=sort([0;ticks(:);-ticks(:)]);
181 set(theaxes, 'Ytick', yFac*asinh(ticks/yScale) ...
182 , 'YtickLabel', cellstr(num2str(ticks,4)))

```

Turn the grid on by default.

```
189 grid on
```

---

## 4 Patches in 2D space

---

## 4.1 configPatches2(): configures spatial patches in 2D

### Section contents

|       |                                                    |     |
|-------|----------------------------------------------------|-----|
| 4.1.1 | If no arguments, then execute an example . . . . . | 139 |
| 4.1.2 | Parse input arguments and defaults . . . . .       | 142 |
| 4.1.3 | The code to make patches . . . . .                 | 144 |
| 4.1.4 | Set ensemble inter-patch communication . . . . .   | 147 |

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)
```

**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  $[Xlim(1), Xlim(2)] \times [Xlim(3), Xlim(4)]$ . 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 ( $\geq 1$ ) 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, . . . ; 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. So either `ratio = 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 must be odd so that there is/are centre-patch micro-grid point/lines in each patch.
- `nEdge` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for macroscale 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.
  - 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.<sup>1</sup>

- ```
207 if nargout==0, global patches, end
      • .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 `.Cwts1`, only for macro-periodic conditions, are the `ordCC` × 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)` × 1 × 1 × 1 × `nPatch(1)` × 1 array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (6D) is 1 × `nSubP(2)` × 1 × 1 × 1 × `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` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.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

```
291 if nargin==0
292 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 (“ \leftrightarrow ” denotes function recursion).

1. `configPatches2`
2. `ode23` integrator \leftrightarrow `patchSys2` \leftrightarrow 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).

```
315 global patches
316 patches = configPatches2(@nonDiffPDE, [-3 3 -2 2], [] ...
317 , [9 7], 0, 0.12, 5 , 'EdgyInt', false);
```

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

```
324 u0 = exp(-patches.x.^2-patches.y.^2);
325 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.

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

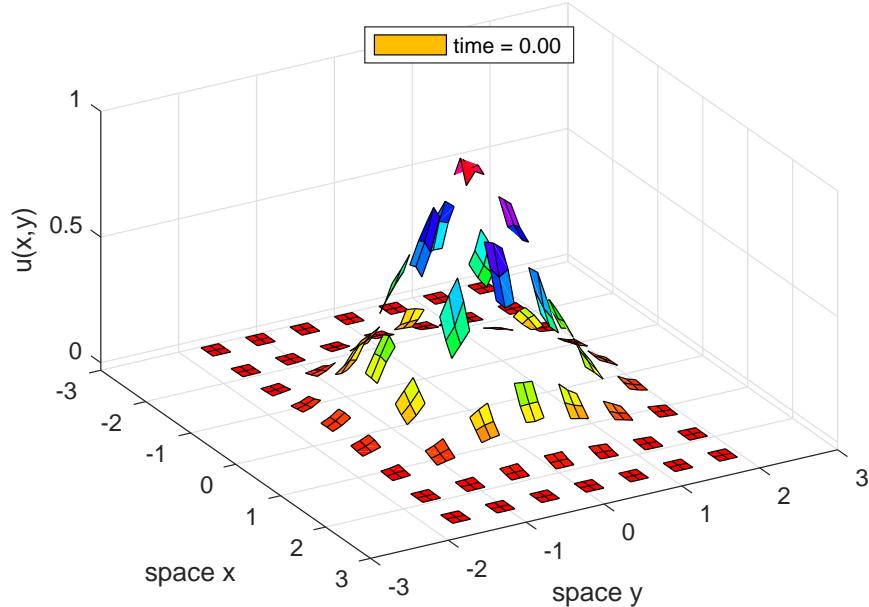
Start by showing the initial conditions of [Figure 4.1](#) while the simulation computes.

```
342 u = reshape(permute(squeeze(u0) ...
343 , [1 3 2 4]), [numel(x) numel(y)]);
344 hsurf = surf(x(:, ), y(:, ), u');
345 axis([-3 3 -3 3 -0.03 1]), view(60,40)
346 legend('time = 0.00', 'Location', 'north')
347 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
348 colormap(hsv)
349 ifOursCf2eps([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.

```
366 disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
367 drawnow
368 if ~exist('OCTAVE_VERSION', 'builtin')
369     [ts,us] = ode23(@patchSys2, linspace(0,2).^2, u0(:));
370 else % octave version is quite slow for me
371     lsode_options('absolute tolerance', 1e-4);
372     lsode_options('relative tolerance', 1e-4);
373     [ts,us] = odeOctave(@patchSys2, [0 1], u0(:));
374 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.

```

382 for i = 1:length(ts)
383     u = patchEdgeInt2(us(i,:));
384     u = reshape(permute(squeeze(u) ...
385         ,[1 3 2 4]), [numel(x) numel(y)]);
386     set(hsurf,'ZData', u');
387     legend(['time = ' num2str(ts(i),'%4.2f')])
388     pause(0.1)
389 end
390 ifOrCf2eps([mfilename 't3'])

```

Upon finishing execution of the example, exit this function.

```

405 return
406 end%if no arguments

```

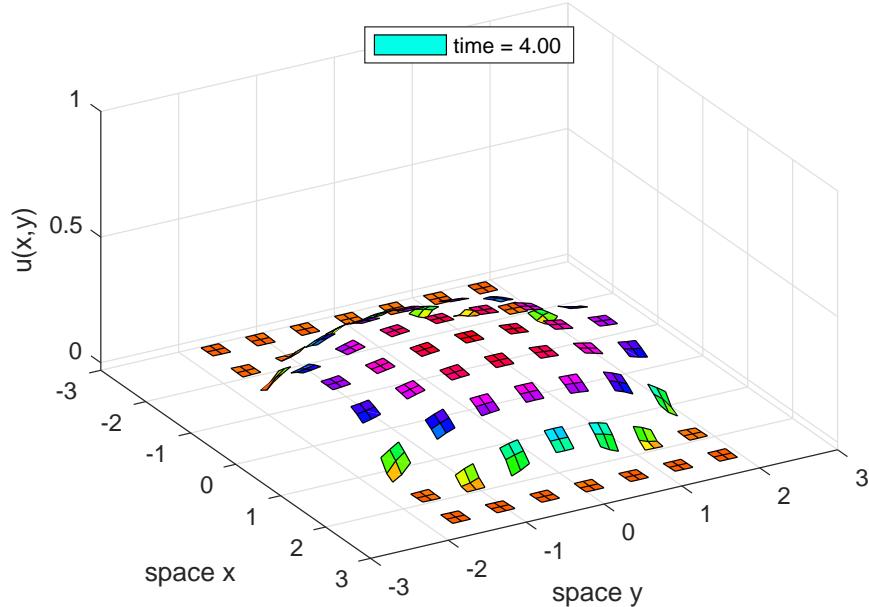
Example of nonlinear diffusion PDE inside patches As a microscale discretisation of $u_t = \nabla^2(u^3)$, code $\dot{u}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i-1,j,k,l}^3) + \frac{1}{\delta y^2}(u_{i,j+1,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i,j-1,k,l}^3)$.

```

13 function ut = nonDiffPDE(t,u,patches)
14     if nargin<3, global patches, end
15     u = squeeze(u); % reduce to 4D
16     dx = diff(patches.x(1:2)); % microgrid spacing
17     dy = diff(patches.y(1:2));
18     i = 2:size(u,1)-1; j = 2:size(u,2)-1; % interior patch points
19     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

```

4.1.2 Parse input arguments and defaults

```

420 p = inputParser;
421 fnValidation = @(f) isa(f, 'function_handle');%test for fn name
422 addRequired(p, 'fun', fnValidation);
423 addRequired(p, 'Xlim', @isnumeric);
424 %addRequired(p, 'Dom'); % nothing yet decided
425 addRequired(p, 'nPatch', @isnumeric);
426 addRequired(p, 'ordCC', @isnumeric);
427 addRequired(p, 'dx', @isnumeric);
428 addRequired(p, 'nSubP', @isnumeric);
429 addParameter(p, 'nEdge', 1, @isnumeric);
430 addParameter(p, 'EdgyInt', false, @islogical);
431 addParameter(p, 'nEnsem', 1, @isnumeric);
432 addParameter(p, 'hetCoeffs', [], @isnumeric);
433 addParameter(p, 'parallel', false, @islogical);
434 %addParameter(p, 'nCore', 1, @isnumeric); % not yet implemented
435 parse(p, fun, Xlim, nPatch, ordCC, dx, nSubP, varargin{:});

```

Set the optional parameters.

```

441 patches.nEdge = p.Results.nEdge;
442 patches.EdgyInt = p.Results.EdgyInt;
443 patches.nEnsem = p.Results.nEnsem;
444 cs = p.Results.hetCoeffs;

```

```

445 patches.parallel = p.Results.parallel;
446 %patches.nCore = p.Results.nCore;

Initially duplicate parameters for both space dimensions as needed.

454 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
455 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
456 if numel(dx)==1, dx = repmat(dx,1,2); end
457 if numel(nSubP)==1, nSubP = repmat(nSubP,1,2); end

Check parameters.

464 assert(Xlim(1)<Xlim(2) ...
465     , 'first pair of Xlim must be ordered increasing')
466 assert(Xlim(3)<Xlim(4) ...
467     , 'second pair of Xlim must be ordered increasing')
468 assert(patches.nEdge==1 ...
469     , 'multi-edge-value interp not yet implemented')
470 assert(all(2*patches.nEdge<nSubP) ...
471     , 'too many edge values requested')
472 %if patches.nCore>1
473 %    warning('nCore>1 not yet tested in this version')
474 %end

```

For compatibility with pre-2023 functions, if parameter Dom is Nan, then we set the ratio to be the value of the so-called dx vector.

```

485 if ~isstruct(Dom), pre2023=isnan(Dom);
486 else pre2023=false; end
487 if pre2023, ratio=dx; dx=nan; end

Default macroscale conditions are periodic with evenly spaced patches.

495 if isempty(Dom), Dom=struct('type','periodic'); end
496 if (~isstruct(Dom))&isnan(Dom), Dom=struct('type','periodic'); end

```

If Dom is a string, then just set type to that string, and subsequently set corresponding defaults for others fields.

```
504 if ischar(Dom), Dom=struct('type',Dom); end
```

We allow different macroscale domain conditions in the different directions. But for the moment do not allow periodic to be mixed with the others (as the interpolation mechanism is different code)—hence why we choose `periodic` be seven characters, whereas the others are eight characters. The different conditions are coded in different rows of `Dom.type`, so we duplicate the string if only one row specified.

```
517 if size(Dom.type,1)==1, Dom.type=repmat(Dom.type,2,1); end
```

Check what is and is not specified, and provide default of zero (Dirichlet boundaries) if no bcOffset specified when needed. Do so for both directions independently.

```
526 patches.periodic=false;
527 for p=1:2
```

```

528 switch Dom.type(p,:)
529 case 'periodic'
530     patches.periodic=true;
531     if isfield(Dom,'bcOffset')
532         warning('bcOffset not available for Dom.type = periodic'), end
533         msg=' not available for Dom.type = periodic';
534         if isfield(Dom,'X'), warning(['X' msg]), end
535         if isfield(Dom,'Y'), warning(['Y' msg]), end
536 case {'equispace','chebyshev'}
537     if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,2); end
538     % for mixed with usergiven, following should still work
539     if numel(Dom.bcOffset)==1
540         Dom.bcOffset= repmat(Dom.bcOffset,2,2); end
541     if numel(Dom.bcOffset)==2
542         Dom.bcOffset=repmat(Dom.bcOffset(:,2,1),2,1); end
543     msg=' not available for Dom.type = equispace or chebyshev';
544     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
545     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
546 case 'usergiven'
547     % if isfield(Dom,'bcOffset')
548     % warning('bcOffset not available for usergiven Dom.type'), end
549     msg=' required for Dom.type = usergiven';
550     if p==1, assert(isfield(Dom,'X'),['X' msg]), end
551     if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
552 otherwise
553     error(['Dom.type ' is unknown Dom.type'])
554 end%switch Dom.type
555 end%for p

```

4.1.3 The code to make patches

First, store the pointer to the time derivative function in the struct.

```
568 patches.fun = fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 or (not yet??) -1. ²

```
578 assert((ordCC>=-1) & (floor(ordCC)==ordCC), ...
579 'ordCC out of allowed range integer>=-1')
```

For odd `ordCC` do interpolation based upon odd neighbouring patches as is useful for staggered grids.

```
586 patches.stag = mod(ordCC,2);
587 assert(patches.stag==0,'staggered not yet implemented??')
588 ordCC = ordCC+patches.stag;
589 patches.ordCC = ordCC;
```

² **ToDo:** Perhaps implement staggered spectral coupling.

Check for staggered grid and periodic case.

```
595 if patches.stag, assert(all(mod(nPatch,2)==0), ...
596     'Require an even number of patches for staggered grid')
597 end
```

Set the macro-distribution of patches Third, set the centre of the patches in the macroscale grid of patches. Loop over the coordinate directions, setting the distribution into \mathbf{Q} and finally assigning to array of corresponding direction.

```
610 for q=1:2
611 qq=2*q-1;
```

Distribution depends upon Dom.type :

```
617 switch Dom.type(q,:)
```

The periodic case is evenly spaced within the spatial domain. Store the size ratio in patches .

```
625 case 'periodic'
626 Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
627 DQ=Q(2)-Q(1);
628 Q=Q(1:nPatch(q))+diff(Q)/2;
629 pEI=patches.EdgyInt;% abbreviation
630 if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-1-pEI)*(2-pEI);
631 else ratio(q) = dx(q)/DQ*(nSubP(q)-1-pEI)/(2-pEI);
632 end
633 patches.ratio=ratio;
```

The equi-spaced case is also evenly spaced but with the extreme edges aligned with the spatial domain boundaries, modified by the offset.

```
642 case 'equispace'
643 Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
644 ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
645 ,nPatch(q));
646 DQ=diff(Q(1:2));
647 width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
648 if DQ<width*0.999999
649 warning('too many equispace patches (double overlapping)')
650 end
```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $Q_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible ‘boundary layers’.³

³ However, maybe overlapping patches near a boundary should be viewed as some sort of spatially analogue of the ‘christmas tree’ of projective integration and its integration to a slow manifold. Here maybe the overlapping patches allow for a ‘christmas tree’ approach to the boundary layers. Needs to be explored??

```

667 case 'chebyshev'
668 halfWidth=dx(q)*(nSubP(q)-1)/2;
669 Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
670 Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
671 % Q = (Q1+Q2)/2-(Q2-Q1)/2*cos(linspace(0,pi,nPatch));

```

Search for total width of ‘boundary layers’ so that in the interior the patches are non-overlapping Chebyshev. But the width for assessing overlap of patches is the following variable `width`.

```

680 width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx(q);
681 for b=0:2:nPatch(q)-2
682 DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
683 if DQmin>width, break, end
684 end
685 if DQmin<width*0.999999
686 warning('too many Chebyshev patches (mid-domain overlap)')
687 end

```

Assign the centre-patch coordinates.

```

693 Q =[ Q1+(0:b/2-1)*width ...
694 (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...
695 Q2+(1-b/2:0)*width ];

```

The user-given case is entirely up to a user to specify, we just ensure it has the correct shape of a row??.

```

704 case 'usergiven'
705 if q==1, Q = reshape(Dom.X,1,[]);
706 else Q = reshape(Dom.Y,1,[]);
707 end%if
708 end%switch Dom.type

```

Assign Q -coordinates to the correct spatial direction. At this stage they are all rows.

```

715 if q==1, X=Q; end
716 if q==2, Y=Q; end
717 end%for q

```

Construct the micro-grids Fourth, construct the microscale grid in each patch. Reshape the grid to be 6D to suit dimensions (micro,Vars,Ens,macro).

```

732 nSubP = reshape(nSubP,1,2); % force to be row vector
733 assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
734 'configPatches2: nSubP must be odd')
735 i0 = (nSubP(1)+1)/2;
736 patches.x = reshape( dx(1)*(-i0+1:i0-1)'+X ...
737 ,nSubP(1),1,1,1,nPatch(1),1);

```

Next the y -direction.

```

743 i0 = (nSubP(2)+1)/2;
744 patches.y = reshape( dx(2)*(-i0+1:i0-1)'+Y ...
745 ,1,nSubP(2),1,1,1,nPatch(2));

```

Pre-compute weights for macro-periodic In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling.⁴

```

756 if patches.periodic
757     ratio = reshape(ratio,1,2); % force to be row vector
758     patches.ratio=ratio;
759     if ordCC>0
760         [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
761         patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
762     end%if
763 end%if patches.periodic

```

4.1.4 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

- the right-edge/centre realisations `1:nEnsem` are to interpolate to left-edge `le`, and
- the left-edge/centre realisations `1:nEnsem` are to interpolate to `re`.

`re` and `li` are ‘transposes’ of each other as `re(li)=le(ri)` are both `1:nEnsem`. Similarly for bottom-edge/centre interpolation to top-edge via `to`, and top-edge/centre interpolation to bottom-edge via `bo`.

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt2()`.

```

790 nE = patches.nEnsem;
791 patches.le = 1:nE; patches.ri = 1:nE;
792 patches.bo = 1:nE; patches.to = 1:nE;

```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more than 3D, then the higher-dimensions are reshaped into the 3rd dimension.

```

804 if ~isempty(cs)
805     [mx,my,nc] = size(cs);
806     nx = nSubP(1); ny = nSubP(2);
807     cs = repmat(cs,nSubP);

```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```

815 if nE==1, patches.cs = cs(1:nx-1,1:ny-1,:); else

```

⁴ **ToDo:** Might sometime extend to coupling via derivative values.

But for `nEnsem` > 1 an ensemble of $m_x m_y$ phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```

824     patches.nEnsem = mx*my;
825     patches.cs = nan(nx-1,ny-1,nc,mx,my);
826     for j = 1:my
827         js = (j:j+ny-2);
828         for i = 1:mx
829             is = (i:i+nx-2);
830             patches.cs(:,:,:,i,j) = cs(is,js,:);
831         end
832     end
833     patches.cs = reshape(patches.cs,nx-1,ny-1,nc,[]);

```

Further, set a cunning left/right/bottom/top realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```

843     le = mod((0:mx-1)+mod(nx-2,mx),mx)+1;
844     patches.le = reshape( le'+mx*(0:my-1) ,[],1);
845     ri = mod((0:mx-1)-mod(nx-2,mx),mx)+1;
846     patches.ri = reshape( ri'+mx*(0:my-1) ,[],1);
847     bo = mod((0:my-1)+mod(ny-2,my),my)+1;
848     patches.bo = reshape( (1:mx)'+mx*(bo-1) ,[],1);
849     to = mod((0:my-1)-mod(ny-2,my),my)+1;
850     patches.to = reshape( (1:mx)'+mx*(to-1) ,[],1);

```

Issue warning if the ensemble is likely to be affected by lack of scale separation.
⁵

```

858 if prod(ratio)*patches.nEnsem>0.9, warning( ...
859 'Probably poor scale separation in ensemble of coupled phase-shifts')
860 scaleSeparationParameter = ratio*patches.nEnsem
861 end

End the two if-statements.

867 end%if-else nEnsem>1
868 end%if not-empty(cs)

```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable patches must only be referenced within an spmd-environment*.⁶

⁵ **ToDo:** Maybe need to justify this and the arbitrary threshold more carefully??

⁶ If subsequently outside `spmd`, then one must use functions like `getfield(patches{1}, 'a')`.

```

887 if patches.parallel
888 % theparpool=gcp()
889 spmd

```

Second, decide which dimension is to be sliced among parallel workers (for the moment, do not consider slicing the ensemble). Choose the direction of most patches, biased towards the last.

```

898 [~,pari]=max(nPatch+0.01*(1:2));
899 patches.codist=codistributor1d(4+pari);

```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the appropriate coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```

909 switch pari
910   case 1, patches.x=codistributed(patches.x,patches.codist);
911   case 2, patches.y=codistributed(patches.y,patches.codist);
912 otherwise
913   error('should never have bad index for parallel distribution')
914 end%switch
915 end%spmd

```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

```

923 else% not parallel
924   if isfield(patches,'codist'), rmfield(patches,'codist'); end
925 end%if-parallel

```

Fin

```

934 end% function

```

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. It assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables ([Section 4.1](#)) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

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

Input

- `u` is a vector/array of length $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ 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 × nEnsem × 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 × 1 × nPatch(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 × nSubP(2) × 1 × 1 × 1 × 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 number 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 $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ and the same dimensions as `u`.

Reshape the fields u as a 6D-array, and sets the edge values from macroscale interpolation of centre-patch values. [Section 4.3](#) describes `patchEdgeInt2()`.

```
98 sizeu = size(u);  
99 u = patchEdgeInt2(u,patches);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero (as `ode15s` chokes on NaNs), then return to the user/integrator as same sized array as input.

```
109 dudt = patches.fun(t,u,patches,varargin{:});  
110 dudt([1 end],:,:,:,:,:) = 0;  
111 dudt(:,[1 end],:,:,:,:,:) = 0;  
112 dudt = reshape(dudt,sizeu);
```

Fin.

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.⁷

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 $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ 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 × **nSubP2** × 1 × 1 × 1 × **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, ...}
 - **.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

⁷ Script `patchEdgeInt2test.m` verifies this code.

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

- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

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

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```
117     if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
118         uclean=@(u) real(u);
119     else uclean=@(u) u;
120     end
```

Determine the sizes of things. Any error arising in the reshape indicates `u` has the wrong size.

```
128 [~,ny,~,~,~,Ny] = size(patches.y);
129 [nx,~,~,~,Nx,~] = size(patches.x);
130 nEnsem = patches.nEnsem;
131 nVars = round(numel(u)/numel(patches.x)/numel(patches.y)/nEnsem);
132 assert(numel(u) == nx*ny*Nx*Ny*nVars*nEnsem ...
133 , 'patchEdgeInt2: input u has wrong size for parameters')
134 u = reshape(u,[nx ny nVars nEnsem Nx Ny]);
```

For the moment assume the physical domain is either macroscale periodic or macroscale rectangle so that the coupling formulas are simplest. These index vectors point to patches and, if periodic, their four immediate neighbours.

```
144 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
145 J=1:Ny; Jp=mod(J,Ny)+1; Jm=mod(J-2,Ny)+1;
```

The centre of each patch (as `nx` and `ny` are odd for centre-patch interpolation) is at indices

```
152 i0 = round((nx+1)/2);
153 j0 = round((ny+1)/2);
```

4.3.1 Periodic macroscale interpolation schemes

```
162 if patches.periodic
```

Get the size ratios of the patches.

```
168 rx = patches.ratio(1);
169 ry = patches.ratio(2);
```

4.3.1.1 Lagrange interpolation gives patch-edge values

Compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Here the domain is macro-periodic.

```
181 ordCC = patches.ordCC;
182 if ordCC>0 % then finite-width polynomial interpolation
```

Interpolate the three directions in succession, in this way we naturally fill-in corner values. Start with x -direction, and give most documentation for that case as the y -direction is essentially the same.

x -normal edge values The patch-edge values are either interpolated from the next-to-edge values, or from the centre-cross values (not the patch-centre value itself as that seems to have worse properties in general). Have not yet implemented core averages.

```
198 if patches.EdgyInt % interpolate next-to-face values
199   U = u([2 nx-1],2:(ny-1),:,:,I,J);
200 else % interpolate centre-cross values
201   U = u(i0,2:(ny-1),:,:,I,J);
202 end;%if patches.EdgyInt
```

Just in case the last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```
210 szU0=size(U); szU0=[szU0 ones(1,6-length(szU0)) ordCC];
```

Use finite difference formulas for the interpolation, so store finite differences ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```
220 if ~patches.parallel, dmu = zeros(szU0); % 7D
221 else dmu = zeros(szU0,patches.codist); % 7D
222 end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2 .

```
228 if patches.stag % use only odd numbered neighbours
229   error('polynomial interpolation not yet for staggered patch coupling')
230 %   dmux(:,:, :, :, I, :, 1) = (Ux(:,:, :, :, Ip, :) + Ux(:,:, :, :, Im, :))/2; % \mu
231 %   dmux(:,:, :, :, I, :, 2) = (Ux(:,:, :, :, Ip, :) - Ux(:,:, :, :, Im, :)); % \delta
232 %   Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
233 %   dmuy(:,:, :, :, J, 1) = (Ux(:,:, :, :, Jp) + Ux(:,:, :, :, Jm))/2; % \mu
234 %   dmuy(:,:, :, :, J, 2) = (Ux(:,:, :, :, Jp) - Ux(:,:, :, :, Jm)); % \delta
235 %   Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
236 else %disp('starting standard interpolation')
237   dmu(:,:, :, :, I, :, 1) = (U(:,:, :, :, Ip, :) ...
238                           - U(:,:, :, :, Im, :))/2; % \mu\delta
239   dmu(:,:, :, :, I, :, 2) = (U(:,:, :, :, Ip, :) ...
240                           - 2*U(:,:, :, :, I, :)) + U(:,:, :, :, Im, :)); % \delta^2
241 end% if patches.stag
```

Recursively take δ^2 of these to form successively higher order centred differences in space.

```

248   for k = 3:ordCC
249     dmu(:,:,(:,:,I,:,k)) =      dmu(:,:,(:,:,Ip,:,k-2)) ...
250     -2*dmu(:,:,(:,:,I,:,k-2)) +dmu(:,:,(:,:,Im,:,k-2));
251   end

```

Interpolate macro-values to be Dirichlet edge values for each patch ([Roberts & Kevrekidis 2007](#), [Bunder et al. 2017](#)), using weights computed in `configPatches2()`. Here interpolate to specified order.

For the case where next-to-edge values interpolate to the opposite edge-values: when we have an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to` and `patches.bo`.

```

266 k=1+patches.EdgyInt; % use centre or two edges
267 u(nx,2:(ny-1),:,patches.ri,I,:)
268 = U(1,:,:,:,:)*(1-patches.stag) ...
269 +sum( shiftdim(patches.Cwtsr(:,1),-6).*dmu(1,:,:,:,:,:,:) ,7);
270 u(1 ,2:(ny-1),:,patches.le,I,:,:) ...
271 = U(k,:,:,:,:,:)*(1-patches.stag) ...
272 +sum( shiftdim(patches.Cwtsl(:,1),-6).*dmu(k,:,:,:,:,:,:) ,7);

```

y-normal edge values Interpolate from either the next-to-edge values, or the centre-cross-line values.

```

282 if patches.EdgyInt % interpolate next-to-face values
283   U = u(:,:,2 ny-1,:,:,:,:,I,J);
284 else % interpolate centre-cross values
285   U = u(:,:,j0,:,:,:,:,I,J);
286 end;%if patches.EdgyInt

```

Adjoin extra dimension for the array of differences.

```
292 szU0=size(U); szU0=[szU0 ones(1,6-length(szU0)) ordCC];
```

Store finite differences ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) in this array.

```

299 if ~patches.parallel, dmu = zeros(szU0); % 7D
300 else dmu = zeros(szU0,patches.codist); % 7D
301 end;%if patches.parallel

```

First compute differences $\mu\delta$ and δ^2 .

```

307 if patches.stag % use only odd numbered neighbours
308   error('polynomial interpolation not yet for staggered patch coupling')
309 else %disp('starting standard interpolation')
310   dmu(:,:,(:,:,J,1)) = (U(:,:,(:,:,Jp)) ...
311                           -U(:,:,(:,:,Jm))/2); \%mu\delta
312   dmu(:,:,(:,:,J,2)) = (U(:,:,(:,:,Jp)) ...
313                           -2*U(:,:,(:,:,J)) +U(:,:,(:,:,Jm))); \%delta^2
314 end% if stag

```

Recursively take δ^2 .

```

320   for k = 3:ordCC
321     dmu(:,:,(:,:,J,k) =      dmu(:,:,(:,:,Jp,k-2) ...
322     -2*dmu(:,:,(:,:,J,k-2) +dmu(:,:,(:,:,Jm,k-2);
323   end

```

Interpolate macro-values using the weights pre-computed by `configPatches2()`. An ensemble of configurations may have cross-coupling.

```

331 k = 1+patches.EdgyInt; % use centre or two edges
332 u(:,:,patches.to,:,:J) ...
333   = U(:,:,1,:,:)*(1-patches.stag) ...
334   +sum( shiftdim(patches.Cwtsr(:,2),-6).*dmu(:,:,1,:,:,:,:) ,7);
335 u(:,:,1,:,:patches.bo,:,:J) ...
336   = U(:,:,k,:,:)*(1-patches.stag) ...
337   +sum( shiftdim(patches.Cwtsl(:,2),-6).*dmu(:,:,k,:,:,:,:) ,7);

```

4.3.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

```
348 else% patches.ordCC<=0, spectral interpolation
```

We interpolate in terms of the patch index, j say, not directly in space. As the macroscale fields are N -periodic in the patch index I , the macroscale Fourier transform writes the centre-patch values as $U_I = \sum_k C_k e^{ik2\pi I/N}$. Then the edge-patch values $U_{I\pm r} = \sum_k C_k e^{ik2\pi N(I\pm r)} = \sum_k C'_k e^{ik2\pi I/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For N patches we resolve ‘wavenumbers’ $|k| < N/2$, so set row vector $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1) - k_{\max}, \dots, -1)$ for even N .

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches2` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```

371 if patches.stag % transform by doubling the number of fields
372 error('staggered grid not yet implemented??')
373 v=nan(size(u)); % currently to restore the shape of u
374 u=cat(3,u(:,:,1:2:nPatch,:),u(:,:,2:2:nPatch,:));
375 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
376 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
377 r=r/2; % ratio effectively halved
378 nPatch=nPatch/2; % halve the number of patches
379 nVars=nVars*2; % double the number of fields
380 else % the values for standard spectral
381   stagShift = 0;
382   iV = 1:nVars;
383 end%if patches.stag

```

Interpolate the two directions in succession, in this way we naturally fill-in edge-corner values. Start with x -direction, and give most documentation for

that case as the other is essentially the same. Need these indices of patch interior.

```
393 ix = 2:nx-1; iy = 2:ny-1;
```

x-normal edge values Now set wavenumbers into a vector at the correct dimension. In the case of even N these compute the + -case for the highest wavenumber zig-zag mode, $k = (0, 1, \dots, k_{\max}, +(k_{\max} + 1) - k_{\max}, \dots, -1)$.

```
406 kMax = floor((Nx-1)/2);
407 kr = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) , -3);
```

Compute the Fourier transform of the centre-cross values. Unless doing patch-edgy interpolation when FT the next-to-edge values. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to` and `patches.bo`.

```
420 if ~patches.EdgyInt
421     Cm = fft( u(i0,iy,:,:,:, :) ,[],5);
422     Cp = Cm;
423 else
424     Cm = fft( u( 2,iy ,:, patches.le,:,:,:) ,[],5);
425     Cp = fft( u(nx-1,iy ,:, patches.ri,:,:,:) ,[],5);
426 end%if ~patches.EdgyInt
```

Now invert the Fourier transforms to complete interpolation. Enforce reality when appropriate.

```
433 u(nx,iy,:,:,:, :) = uclean( ifft( ...
434     Cm.*exp(1i*(stagShift+kr)) ,[],5) );
435 u( 1,iy,:,:,:, :) = uclean( ifft( ...
436     Cp.*exp(1i*(stagShift-kr)) ,[],5) );
```

y-normal edge values Set wavenumbers into a vector.

```
446 kMax = floor((Ny-1)/2);
447 kr = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMax,Ny)-kMax) , -4);
```

Compute the Fourier transform of the patch values on the centre-lines for all the fields.

```
454 if ~patches.EdgyInt
455     Cm = fft( u(:,j0,:,:,:, :) ,[],6);
456     Cp = Cm;
457 else
458     Cm = fft( u(:,2 ,:, patches.bo,:,:,:) ,[],6);
459     Cp = fft( u(:,ny-1 ,:, patches.to,:,:,:) ,[],6);
460 end%if ~patches.EdgyInt
```

Invert the Fourier transforms to complete interpolation.

```

466 u(:,:,,:,::,:,:) = uclean( ifft( ...
467     Cm.*exp(1i*(stagShift+kr)) ,[],6) );
468 u(:,:,1,:,:,:,:) = uclean( ifft( ...
469     Cp.*exp(1i*(stagShift-kr)) ,[],6) );
475 end% if ordCC>0 else, so spectral

```

4.3.2 Non-periodic macroscale interpolation

```

486 else% patches.periodic false
487 assert(~patches.stag, ...
488 'not yet implemented staggered grids for non-periodic')

```

Determine the order of interpolation `px` and `py` (potentially different in the different directions!), and hence size of the (forward) divided difference tables in `F` (7D) for interpolating to left/right, and top/bottom edges. Because of the product-form of the patch grid, and because we are doing *only* either edgy interpolation or cross-patch interpolation (*not* just the centre patch value), the interpolations are all 1D interpolations.

```

502 if patches.ordCC<1
503     px = Nx-1; py = Ny-1;
504 else px = min(patches.ordCC,Nx-1);
505     py = min(patches.ordCC,Ny-1);
506 end
507 ix=2:nx-1; iy=2:ny-1; % indices of edge 'interior' (ix n/a)

```

4.3.2.1 x -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble. For `EdgyInt`, the ‘reversal’ of the next-to-edge values are because their values are to interpolate to the opposite edge of each patch.⁸

```

520 F = nan(patches.EdgyInt+1,ny-2,nVars,nEnsem,Nx,Ny,px+1);
521 if patches.EdgyInt % interpolate next-to-edge values
522     F(:,:,:,:,1,:,:) = u([nx-1 2],iy,:,:,:,:,:);
523     X = patches.x([nx-1 2],:,:,:,:,:,:);
524 else % interpolate mid-patch cross-patch values
525     F(:,:,:,:,1,:,:) = u(i0,iy,:,:,:,:,:);
526     X = patches.x(i0,:,:,:,:,:);
527 end%if patches.EdgyInt

```

Form tables of divided differences Compute tables of (forward) divided differences (e.g., [Wikipedia 2022](#)) for every variable, and across ensemble, and for left/right edges. Recursively find all divided differences.

```

538 for q = 1:px
539     i = 1:Nx-q;
540     F(:,:,:,:,i,:,q+1) ...
541         = (F(:,:,:,:,i+1 ,:,q)-F(:,:,:,:,i,:,q)) ...

```

⁸ **ToDo:** Have no plans to implement core averaging as yet.

```

542     ./ (X(:,:, :, :, i+q, :) - X(:,:, :, i, :));
543 end

```

Interpolate with divided differences Now interpolate to find the edge-values on left/right edges at X_{edge} for every interior Y .

```
552 Xedge = patches.x([1 nx], :, :, :, :, :);
```

Code Horner's recursive evaluation of the interpolation polynomials. Indices i are those of the left edge of each interpolation stencil, because the table is of forward differences. This alternative: the case of order p_x and p_y interpolation across the domain, asymmetric near the boundaries of the rectangular domain.

```

563 i = max(1,min(1:Nx,Nx-ceil(px/2))-floor(px/2));
564 Uedge = F(:,:, :, i, :, px+1);
565 for q = px:-1:1
566     Uedge = F(:,:, :, i, :, q)+(Xedge-X(:,:, :, i+q-1, :)).*Uedge;
567 end

```

Finally, insert edge values into the array of field values, using the required ensemble shifts.

```

575 u(1 ,iy,: ,patches.le,:,: ) = Uedge(1,:,:,:, :, :);
576 u(nx, iy,: ,patches.ri,:,: ) = Uedge(2,:,:,:, :, :);

```

4.3.2.2 y -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble.

```

585 F = nan(nx, patches.EdgyInt+1, nVars, nEnsem, Nx, Ny, py+1);
586 if patches.EdgyInt % interpolate next-to-edge values
587     F(:,:, :, :, :, 1) = u(:, [ny-1 2], :, :, :, :);
588     Y = patches.y(:, [ny-1 2], :, :, :, :);
589 else % interpolate mid-patch cross-patch values
590     F(:,:, :, :, :, 1) = u(:, j0, :, :, :, :);
591     Y = patches.y(:, j0, :, :, :, :);
592 end;

```

Form tables of divided differences.

```

598 for q = 1:py
599     j = 1:Ny-q;
600     F(:,:, :, :, :, j, q+1) ...
601     = (F(:,:, :, :, :, j+1 , q)-F(:,:, :, :, :, j, q)) ...
602     ./ (Y(:,:, :, :, :, j+q) - Y(:,:, :, :, :, j));
603 end

```

Interpolate to find the edge-values on top/bottom edges Y_{edge} for every x .

```
610 Yedge = patches.y(:, [1 ny], :, :, :, :);
```

Code Horner's recursive evaluation of the interpolation polynomials. Indices j are those of the bottom edge of each interpolation stencil, because the table is of forward differences.

```

619     j = max(1,min(1:Ny,Ny-ceil(py/2))-floor(py/2));
620     Uedge = F(:,:,(:,:,j,py+1));
621     for q = py:-1:1
622         Uedge = F(:,:,(:,:,j,q)+(Yedge-Y(:,:,(:,:,j+q-1)).*Uedge);
623     end

```

Finally, insert edge values into the array of field values, using the required ensemble shifts.

```

630     u(:,1 ,:,patches.bo,:,:)= Uedge(:,1,:,:,:);
631     u(:,ny,:,:patches.to,:,:)= Uedge(:,2,:,:,:);

```

4.3.2.3 Optional NaNs for safety

We want a user to set outer edge values on the extreme patches according to the microscale boundary conditions that hold at the extremes of the domain. Consequently, override their computed interpolation values with NaN.

```

643     u( 1,:,:,:, 1,:) = nan;
644     u(nx,:,:,:,Nx,:) = nan;
645     u(:, 1,:,:,:, 1) = nan;
646     u(:,ny,:,:,:,Ny) = nan;

```

End of the non-periodic interpolation code.

```
653 end%if patches.periodic else
```

Fin, returning the 6D array of field values with interpolated edges.

```
662 end% function patchEdgeInt2
```

4.4 wave2D: example of a wave on patches in 2D

Section contents

4.4.1	Check on the linear stability of the wave PDE	161
4.4.2	Execute a simulation	162
4.4.3	wavePDE(): Example of simple wave PDE inside patches	163

For $u(x, y, t)$, test and simulate the simple wave PDE in 2D space:

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u.$$

This script shows one way to get started: a user's script may have the following three steps (left-right arrows denote function recursion).

1. configPatches2
2. ode15s integrator \leftrightarrow patchSys2 \leftrightarrow wavePDE
3. process results

Establish the global data struct `patches` to interface with a function coding the wave PDE: to be solved on 2π -periodic domain, with 9×9 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25 (big enough for visualisation), and with a 5×5 micro-grid within each patch.

```
35 global patches
36 nSubP = 5;
37 nPatch = 9;
38 configPatches2(@wavePDE, [-pi pi], nan, nPatch, 0, 0.25, nSubP);
```

4.4.1 Check on the linear stability of the wave PDE

Construct the systems Jacobian via numerical differentiation. Set a zero equilibrium as basis. Then find the indices of patch-interior points as the only ones to vary in order to construct the Jacobian.

```
51 disp('Check linear stability of the wave scheme')
52 uv0 = zeros(nSubP,nSubP,2,1,nPatch,nPatch);
53 uv0([1 end],:,:,:,:,:) = nan;
54 uv0(:,[1 end],:,:,:,:,:) = nan;
55 i = find(~isnan(uv0));
```

Now construct the Jacobian. Since this is a *linear* wave PDE, use large perturbations.

```
62 small = 1;
63 jac = nan(length(i));
64 sizeJacobian = size(jac)
65 for j = 1:length(i)
66     uv = uv0(:);
67     uv(i(j)) = uv(i(j))+small;
```

```

68     tmp = patchSys2(0,uv)/small;
69     jac(:,j) = tmp(i);
70 end

```

Now explore the eigenvalues a little: find the ten with the biggest real-part; if these are small enough, then the method may be good.

```

78 evals = eig(jac);
79 nEvals = length(evals)
80 [~,k] = sort(-abs(real(evals)));
81 evalsWithBiggestRealPart = evals(k(1:10))
82 if abs(real(evals(k(1))))>1e-4
83     warning('eigenvalue failure: real-part > 1e-4')
84     return, end

```

Check that the eigenvalues are close to true waves of the PDE (not yet the micro-discretised equations).

```

91 kwave = 0:(nPatch-1)/2;
92 freq = sort(reshape(sqrt(kwave.^2+kwave.^2),1,[]));
93 freq = freq(diff([-1 freq])>1e-9);
94 freqerr = [freq; min(abs(imag(evals)-freq))]

```

4.4.2 Execute a simulation

Set a Gaussian initial condition using auto-replication of the spatial grid: here u_0 and v_0 are in the form required for computation: $n_x \times n_y \times 1 \times 1 \times N_x \times N_y$.

```

109 u0 = exp(-patches.x.^2-patches.y.^2);
110 v0 = zeros(size(u0));

```

Initiate a plot of the simulation using only the microscale values interior to the patches: set x and y -edges to `nan` to leave the gaps. Start by showing the initial conditions of [Figure 4.1](#) while the simulation computes. To mesh/surf plot we need to ?? ‘transpose’ to size $n_x \times N_x \times n_y \times N_y$, then reshape to size $n_x \cdot N_x \times n_y \cdot N_y$.

```

122 x = squeeze(patches.x); y = squeeze(patches.y);
123 x([1 end],:) = nan; y([1 end],:) = nan;
124 u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);
125 usurf = surf(x(:,y(:,u'));
126 axis([-3 3 -3 3 -0.5 1]), view(60,40)
127 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
128 legend('time = 0','Location','north')
129 colormap(hsv)
130 drawnow
131 ifOrCf2eps([mfilename 'ic'])

```

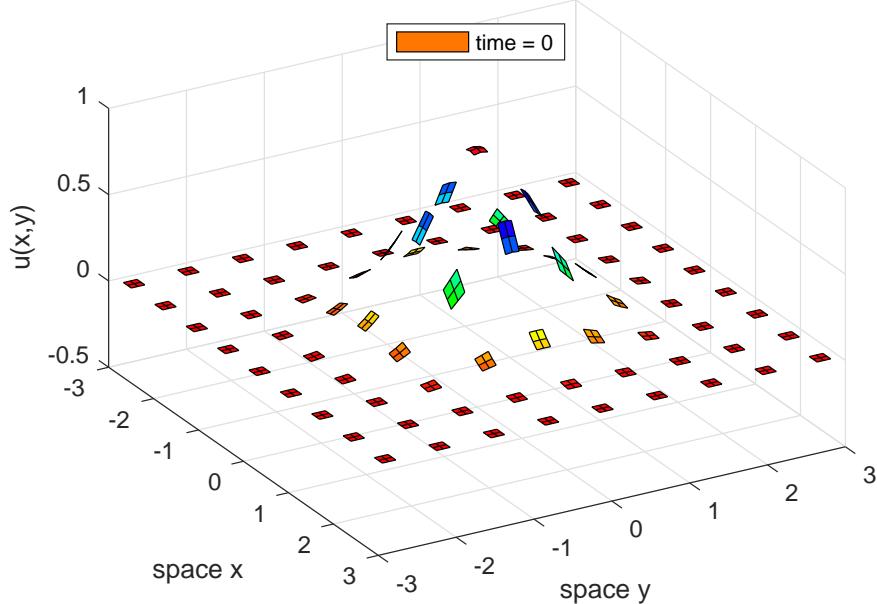
Integrate in time using standard functions.

```

144 disp('Wait while we simulate u_t=v, v_t=u_xx+u_yy')
145 uv0 = cat(3,u0,v0);
146 if ~exist('OCTAVE_VERSION','builtin')

```

Figure 4.3: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to the simple wave PDE: [Figure 4.4](#) plots the computed field at time $t = 2$.



```

147 [ts,uv0] = ode23( @patchSys2,[0 6],uv0(:));
148 else % octave version is slower
149 [ts,uv0] = odeOcts(@patchSys2,linspace(0,6),uv0(:));
150 end

```

Animate the computed simulation to end with [Figure 4.4](#). Because of the very small time-steps, subsample to plot at most 100 times.

```

158 di = ceil(length(ts)/100);
159 for i = [1:di:length(ts)-1 length(ts)]
160   uv = patchEdgeInt2(uvs(i,:));
161   uv = reshape(permute(uv,[1 5 2 6 3 4]), [numel(x) numel(y) 2]);
162   set(usrurf,'ZData', uv(:,:,1));
163   legend(['time = ', num2str(ts(i),2)])
164   pause(0.1)
165 end
166 if0urCf2eps([mfilename 't' num2str(ts(end))])

```

4.4.3 wavePDE(): Example of simple wave PDE inside patches

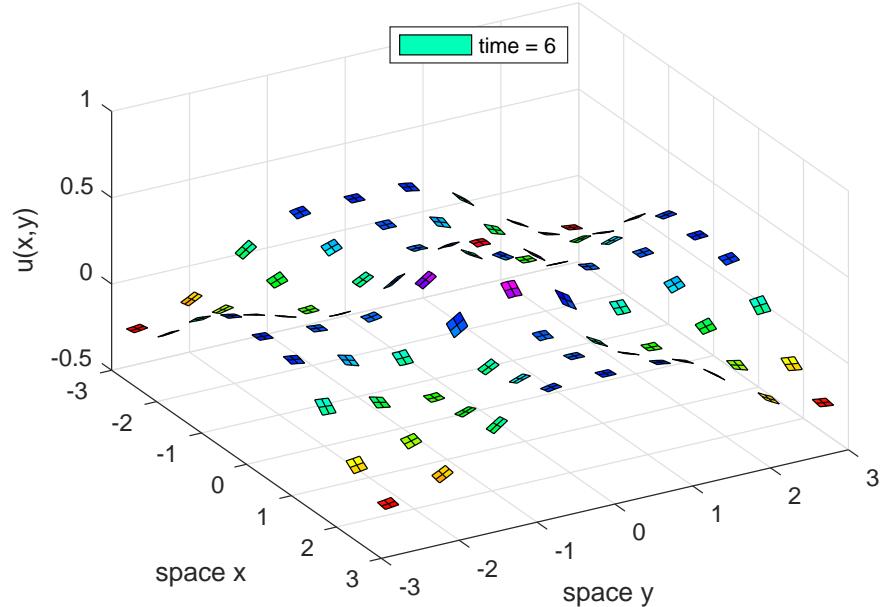
As a microscale discretisation of $u_{tt} = \nabla^2(u)$, so code $\dot{u}_{ijkl} = v_{ijkl}$ and $\ddot{v}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l} - 2u_{i,j,k,l} + u_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u_{i,j+1,k,l} - 2u_{i,j,k,l} + u_{i,j-1,k,l})$.

```

14 function uvt = wavePDE(t,uv,patches)
15   dx = diff(patches.x(1:2));
16   dy = diff(patches.y(1:2)); % microscale spacing
17   i = 2:size(uv,1)-1;
18   j = 2:size(uv,2)-1; % interior patch-points
19   uvt = nan+uv; % preallocate storage
20   uvt(i,j,1,:) = uv(i,j,2,:);

```

Figure 4.4: field $u(x, y, t)$ at time $t = 6$ of the patch scheme applied to the simple wave PDE with initial condition in Figure 4.3.



```

21     uvt(i,j,2,:) = diff(uv(:,j,1,:),2,1)/dx^2 ...
22             +diff(uv(i,:,:1,:),2,2)/dy^2;
23 end

10 function [ts,xs] = ode0cts(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

```

4.5 homoDiffEdgy2: computational homogenisation of a 2D diffusion via simulation on small patches

This section extends to 2D the 1D code discussed in [Section 3.5](#). First set random heterogeneous diffusivities of random period in each of the two directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
23 mPeriod = randi([2 3],1,2)
24 cHetr = exp(1*randn([mPeriod 2]));
25 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 2D we appear to get only real eigenvalues by using edgy interpolation. What happens for non-edgy interpolation is unknown.

```
36 edgyInt = true;
37 nEnsem = 1 %prod(mPeriod) % or just set one
38 if nEnsem==1% use more patches
39     nPatch = [9 9]
40     nSubP = (2-edgyInt)*mPeriod+1+edgyInt
41 else % when nEnsem>1 use fewer patches
42     nPatch = [5 5]
43     nSubP = mPeriod+randi([1 4],1,2) % +2 is decoupled
44 end
45 ratio = 0.2+0.2*rand(1,2)
46 configPatches2(@heteroDiff2,[-pi pi -pi pi],nan,nPatch ...
47 ,0, ratio, nSubP , 'EdgyInt',edgyInt , 'nEnsem',nEnsem ...
48 , 'hetCoeffs',cHetr );
```

Simulate Set initial conditions of a simulation, replicated for each in the ensemble.

```
58 global patches
59 u0 = 0.8*cos(patches.x).*sin(patches.y) ...
60     +0.1*randn([nSubP,1,1,nPatch]);
61 u0 = repmat(u0,1,1,1,nEnsem,1,1);
```

Integrate using standard integrators, unevenly spaced in time to better display transients.

```
68 if ~exist('OCTAVE_VERSION','builtin')
69     [ts,us] = ode23(@patchSys2, 0.3*linspace(0,1).^2, u0(:));
70 else % octave version
71     [ts,us] = odeOctave(@patchSys2, 0.3*linspace(0,1).^2, u0(:));
72 end
```

Plot the solution as an animation over time.

```
79 if ts(end)>0.099, disp('plot animation of solution field')
80 figure(1), clf, colormap(hsv)
```

Get spatial coordinates and pad them with NaNs to separate patches.

```
87 x = squeeze(patches.x); y = squeeze(patches.y);
88 x(end+1,:)=nan; y(end+1,:)=nan; % pad with nans
```

For every time step draw the surface and pause for a short display.

```
95 for i = 1:length(ts)
```

Get the row vector of data, form into the 6D array via the interpolation to the edges, then pad with Nans between patches, and reshape to suit the surf function.

```
103 u = squeeze( mean(patchEdgeInt2(us(i,:),4)) );
104 u(end+1,:,:,:)=nan; u(:,end+1,:,:,:)=nan;
105 u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
```

If the initial time then draw the surface with labels, otherwise just update the surface data.

```
112 if i==1
113     hsurf = surf(x(:,y(:,u')); view(60,40)
114     axis([-pi pi -pi pi -1 1]), caxis([-1 1])
115     xlabel('x'), ylabel('y'), zlabel('u(x,y)')
116 else set(hsurf,'ZData', u');
117 end
118 legend(['time = ' num2str(ts(i),2)],'Location','north')
119 pause(0.05)
```

finish the animation loop and if-plot.

```
125 end%for over time
126 end%if-plot
```

4.5.1 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. Except here use a small ratio as we do not plot.

```
142 ratio = [0.1 0.1]
143 nLeadEvals=prod(nPatch)+max(nPatch);
144 leadingEvals=[];
```

Evaluate eigenvalues for spectral as the base case for polynomial interpolation of order 2, 4,

```
152 maxords=10;
153 for ord=0:2:maxords
    ord=ord
```

Configure with same parameters, then because they are reset by this configuration, restore coupling.

```

161 configPatches2(@heteroDiff2,[-pi pi -pi pi],nan,nPatch ...
162 ,ord, ratio, nSubP, 'EdgyInt', edgyInt, 'nEnsem', nEnsem ...
163 , 'hetCoeffs', cHetr);

```

Find which elements of the 6D array are interior micro-grid points and hence correspond to dynamical variables.

```

170 u0 = zeros([nSubP,1,nEnsem,nPatch]);
171 u0([1 end],:,:) = nan;
172 u0(:,[1 end],:) = nan;
173 i = find(~isnan(u0));

```

Construct the Jacobian of the scheme as the matrix of the linear transformation, obtained by transforming the standard unit vectors.

```

181 jac = nan(length(i));
182 sizeJacobian = size(jac)
183 for j = 1:length(i)
184     u = u0(:)+(i(j)==(1:numel(u0))');
185     tmp = patchSys2(0,u);
186     jac(:,j) = tmp(i);
187 end

```

Test for symmetry, with error if we know it should be symmetric.

```

194 notSymmetric=norm(jac-jac')
195 if edgyInt, assert(notSymmetric<1e-7,'failed symmetry')
196 elseif notSymmetric>1e-7, disp('failed symmetry')
197 end

```

Find all the eigenvalues (as `eigs` is unreliable).

```

203 if edgyInt, [evecs,evals] = eig((jac+jac')/2,'vector');
204 else evals = eig(jac);
205 end
206 biggestImag=max(abs(imag(evals)));
207 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.

```

217 [~,k] = sort(-real(evals));
218 evals=evals(k); evecs=evecs(:,k);
219 if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
220 if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
221 else % find evec closest to that of each leading spectral
222     [~,k]=max(abs(evecs'*evec0));
223     evals=evals(k); % sort in corresponding order
224 end
225 leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
226 end

```

```

227 disp('      spectral      quadratic      quartic sixth-order ...')
228 leadingEvals=leadingEvals

Plot the errors in the eigenvalues using the spectral ones as accurate. Only
plot every second, iEv, as all are repeated eigenvalues.

237 if maxords>2
238     iEv=2:2:12;
239     figure(2);
240     err=abs(leadingEvals-leadingEvals(:,1)) ...
241         ./(1e-7+abs(leadingEvals(:,1)));
242     semilogy(2:2:maxords,err(iEv,2:end)', 'o:');
243     xlabel('coupling order')
244     ylabel('eigenvalue relative error')
245     leg=legend( ...
246         strcat('$',num2str(real(leadingEvals(iEv,1)), '%.4f'), '$') ...
247         , 'Location', 'northeastoutside');
248     if ~exist('OCTAVE_VERSION','builtin')
249         title(leg,'eigenvalues'), end
250     legend boxoff
251 end%if-plot

```

4.5.2 heteroDiff2(): heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 6D input arrays u , x , and y (via edge-value interpolation of `patchSys2`, [Section 4.2](#)), computes the time derivative (3.1) at each point in the interior of a patch, output in ut . The two 2D array of diffusivities, c_{ij}^x and c_{ij}^y , have previously been stored in `patches.cs` (3D).

```

19 function ut = heteroDiff2(t,u,patches)
20     dx = diff(patches.x(2:3)); % x space step
21     dy = diff(patches.y(2:3)); % y space step
22     ix = 2:size(u,1)-1; % x interior points in a patch
23     iy = 2:size(u,2)-1; % y interior points in a patch
24     ut = nan+u;           % preallocate output array
25     ut(ix,iy,:,:,:,:,:) ...
26     = diff(patches.cs(:,iy,1,:).*diff(u(:,iy,:,:,:,:),1,1)/dx^2 ...
27         +diff(patches.cs(ix,:,2,:).*diff(u(ix,:,:,:,:,:),1,2),1,2)/dy^2;
28 end% function

```

Fin.

4.6 homoDiffSoln2: steady state of a 2D heterogeneous diffusion via small patches

Here we find the steady state $u(x, y)$ to the heterogeneous PDE

$$u_t = \vec{\nabla} \cdot [c(x, y) \vec{\nabla} u] - u + f, \quad \text{for } f = 100 \sin(\pi x) \sin(\pi y).$$

The heterogeneous diffusion c varies over two orders of magnitude in small space distance ϵ . I include $-u$ in the PDE to ensure a steady state with periodic BCs.

[Section 4.6.2](#) gives a function that we invoke to explore the errors in the patch scheme solution. The spectral patch scheme is essentially exact.

[Biezemans et al. \(2022\)](#) discussed an example homogenisation in 2D with heterogeneity of period $\epsilon := \pi/150$ in both directions. Ensure integer multiple of heterogeneity periods in the domain, and initially use three times bigger ϵ .

```
35 epsilon = 1/round(50/pi)
```

[Biezemans et al. \(2022\)](#) choose microscale mesh spacing of $1/1024$, so the number of micro-grid points in one period would be 1024ϵ . But *initially* use less.

```
44 mPeriod = round(128*epsilon) %round(1024*epsilon)
```

So the migro-grid spacing is exactly

```
50 dx = epsilon/mPeriod
```

Diffusivities Now form one period of the heterogeneity diffusivities. [Biezemans et al. \(2022\)](#) used $c = 1 + 100 \cos^2(\pi x/\epsilon) \sin^2(\pi y/\epsilon)$. Need to shift phases of the diffusivity by half-micro-grid for diffusivities in each direction to form two diffusivity matrices on the microscale lattice. Variables h, v represent $\pi x/\epsilon$ or $\pi y/\epsilon$.

```
64 cHetr=[];
65 v=pi*( 1:mPeriod)/mPeriod;
66 h=pi*(0.5:mPeriod)/mPeriod;
67 cHetr(:,:,1) = 1+100*cos(h').^2*sin(v).^2;
68 cHetr(:,:,2) = 1+100*cos(v').^2*sin(h).^2;
```

Plot surfaces of the diffusivity.

```
74 figure(2),surf(h/pi,v/pi,cHetr(:,:,2))
75 hold on, surf(v/pi,h/pi,cHetr(:,:,1))
76 hold off, alpha 0.5, drawnow
```

Patch configuration As is common, [Biezemans et al. \(2022\)](#) implemented zero-Dirichlet BCs on $(0, 1)^2$. Here these are more-or-less encompassed by implementing periodic BCs on $(-1, 1)^2$. Initially use 8×8 patches to have 4×4 patches in $(0, 1)^2$, which then have patch spacing H .

```

89 nPatch = [8 8]
90 H = 2./nPatch
91 HepsilonRatio = H/epsilon

```

Best when each patch spans an integral number of periods plus one grid step.
The smallest such patches are

```

98 nSubP = [1 1]*mPeriod+2

```

Consequently, the ratio of space computed on, to the space in the domain is the product of the following ratios in each direction, namely about 8% here.

```

106 ratio = ((nSubP-2)*dx)./H

```

Specify spectral interpolation. The edgy interpolation is self-adjoint ([Bunder et al. 2020](#)) leading to a symmetric matrix problem.

```

114 configPatches2(@hetDiffForce2, [-1 1 -1 1], nan, nPatch ...
115 , 0, ratio, nSubP, 'EdgyInt', true ...
116 , 'hetCoeffs', cHetr );

```

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.

```

128 global patches i
129 u0 = zeros([nSubP,1,1,nPatch]);
130 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;
131 i = find(~isnan(u0));
132 nVars = numel(i)

```

Solve by iteration. Could use `fsolve` for nonlinear problems, but for linear it is much faster to use Conjugate-Gradient algorithm. `gmres` is competitive, but appears to take twice as long.

```

142 tic;
143 if 0, uSoln=fsolve(@theRes,u0(i));

```

The above is for nonlinear PDEs. For linear PDEs, determine the RHS vector, and make a function that computes the matrix vector product.

```

151 else
152     maxIt = ceil(nVars/10);
153     rhsb = theRes(u0(i));
154     uSoln = pcg(@(u) rhsb-theRes(u),rhsb,1e-9,maxIt);
155 end
156 solnTime = toc

```

Store the solution into the patches, and give magnitudes.

```

162 u0(i) = uSoln;
163 normSoln = norm(uSoln)
164 normResidual = norm(theRes(uSoln))

```

Draw solution profile First reshape arrays to suit 2D space surface plots.

```

172 figure(1), clf, colormap(hsv)
173 x = squeeze(patches.x); y = squeeze(patches.y);
174 u = reshape(permute(squeeze(u0),[1 3 2 4]), [numel(x) numel(y)]);

```

Draw the patch solution surface in the positive quadrant, with edge-values omitted as already NaN by not bothering to interpolate them.

```

182 surf(x(:,y(:,u')); view(60,40)
183 maxu = ceil(max(u(:))*10)/10;
184 axis([0 1 0 1 0 maxu]), caxis([0 maxu])
185 xlabel('x'), ylabel('y'), zlabel('u(x,y)')

```

Assess errors in the patch scheme Invoke the function with desired interpolation: 0, spectral; 2, 4, ..., polynomial.

```
194 errorsPatchScheme(0)
```

4.6.1 Microscale discretisation inside patches

hetDiffForce2(): heterogeneous diffusion PDE This function, based upon [Section 4.5.2](#), 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 . The two 2D array of diffusivities, c_{ij}^x and c_{ij}^y , are stored in `patches.cs` (3D).

```

213 function ut = hetDiffForce2(t,u,patches)
214     dx = diff(patches.x(2:3)); % x space step
215     dy = diff(patches.y(2:3)); % y space step
216     ix = 2:size(u,1)-1; % x interior points in a patch
217     iy = 2:size(u,2)-1; % y interior points in a patch
218     ut = nan+u;           % preallocate output array
219     fu = -u+100*sin(pi*patches.x).*sin(pi*patches.y);
220     ut(ix,iy,:,:,:, :) ...
221     = diff(patches.cs(:,iy,1).*diff(u(:,iy,:,:,:,1),1),1)/dx^2 ...
222     +diff(patches.cs(ix,:,2).*diff(u(ix,:,:,:,1,2),1,2),1,2)/dy^2 ...
223     +fu(ix,iy,:,:,:, :);
224 end% function

```

theRes(): function to zero This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system, and returns the vector of patch-interior time derivatives.

```

236 function f=theRes(u)
237 global i patches
238 v=nan(size(patches.x+patches.y));
239 v(i)=u;
240 f=patchSys2(0,v(:,),patches);
241 f=f(i);
242 end

```

4.6.2 Function to explore errors in the patch scheme

We find the spectral interpolation patch scheme accurate to essentially zero errors, namely errors less than 10^{-10} . Non-spectral interpolation has errors that decrease roughly like expected power of patch spacing.

The single argument `ord` is 0 for spectral interpolation and 2, 4, ... for corresponding polynomial interpolation schemes.

```
262 function errorsPatchScheme(ord)
263 warning('Assessing errors via varying number of patches')
```

Use a hierarchy of cases with increasing number of patches—the number increasing by 3^2 from one level to the next in the hierarchy. Then the higher resolution patches precisely contain the lower resolution cases. The case when index `k=kMax` corresponds to the full-domain solution. [Biezemans et al. \(2022\)](#) use heterogeneity of period $\epsilon := \pi/150 \approx 0.021$ in both directions, here with `kMax=3` use $\epsilon \approx 0.037$. Ensure integer multiple of heterogeneity periods in the full domain.

```
279 kMax = 3
280 epsilon = 1/3^kMax
```

[Biezemans et al. \(2022\)](#) choose microscale mesh spacing of 1/1024, so their number of micro-grid points in one period is $1024\epsilon \approx 21$. But here use less because less is plenty enough—the issue is the accuracy of the patch scheme to whatever micro-grid system is given, *not* the accuracy of the micro-grid system to the PDE.

```
292 mPeriod = 9
```

So the migro-grid spacing is

```
298 dx = epsilon/mPeriod
```

Diffusivities Now form one period of the heterogeneity diffusivities exactly as in above code.

```
307 cHetr=[];
308 v=pi*( 1:mPeriod)/mPeriod;
309 h=pi*(0.5:mPeriod)/mPeriod;
310 cHetr(:,:,1) = 1+100*cos(h').^2*sin(v).^2;
311 cHetr(:,:,2) = 1+100*cos(v').^2*sin(h).^2;
```

Loop over different patch spacings

```
319 for k=1:kMax
320 nPatch = [2 2]*3^k
```

Patch configuration Zero-Dirichlet BCs on $(0, 1)^2$ are more-or-less encompassed by implementing periodic BCs on $(-1, 1)^2$.

```
330 H = 2./nPatch
331 HepsilonRatio = H/epsilon
```

Best when each patch spans an integral number of periods plus one grid step. The smallest such patches are

```
338 nSubP = [1 1]*mPeriod+2
```

Consequently, the ratios of space computed on is the following. The case $k=kMax$ gives a ratio of precisely one that characterises a full-domain problem.

```
346 ratio = ((nSubP-2)*dx)./H
```

The edgy interpolation leads to a symmetric matrix problem (Bunder et al. 2020).

```
353 configPatches2(@hetDiffForce2,[-1 1 -1 1],nan,nPatch ...
354     ,ord,ratio,nSubP , 'EdgyInt',true ...
355     , 'hetCoeffs',cHetr );
```

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.

```
367 global patches i
368 u0 = zeros([nSubP,1,1,nPatch]);
369 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;
370 i = find(~isnan(u0));
371 nVars = numel(i)
```

For this linear problem it is fast to solve with the Conjugate-Gradient algorithm. Determine the RHS vector, and use a function that computes the matrix vector product.

```
380 tic
381 rhsb = theRes(u0(i));
382 uSoln = pcg(@(u) rhsb-theRes(u),rhsb,1e-6,999);
383 solnTime = toc
```

Store the solution into the patches, and trace magnitudes.

```
389 u0(i) = uSoln;
390 normSoln = norm(uSoln)
391 normResidual = norm(theRes(uSoln))
```

End loop over different patch spacings Store 4D field values in cell array for post-processing.

```
399 us{k}=squeeze(u0);
400 end%for
```

Compare errors across cases There are nine patches common to all grids (36 if one counts all quadrants), indexed by the following patch indices.

```
410 disp('**** Relate errors for different patch spacing ****')
411 if ord==0, disp('**** Spectral interpolation between patches')
412 else disp(['**** Polynomial interpolation, order ' num2str(ord)])
```

```

413 end
414 i=2:nSubP-1;
415 I{1}=1:3;
416 for k=2:kMax, I{k}=3*I{k-1}-1; end

```

Determine errors by computing difference between patch schemes: the final patch scheme is a full-domain solution and hence ‘exact’. Look at the RMS error in each of the patches. Find the overall error for each patch, their ratios, and the rough order of decrease.

```

426 rmsError=[]; errorRatios=[]; orderInH=[];
427 for k=1:kMax-1
428     error{k}=us{k}(i,i,I{k},I{k})-us{kMax}(i,i,I{kMax},I{kMax});
429     rmsError(:,:,k)=squeeze(rms(rms(error{k})));
430     if (k>1)&(ord>0)
431         errorRatios(:,:,k-1)=rmsError(:,:,k)./rmsError(:,:,k-1);
432         orderInH(:,:,k-1)=-log(errorRatios(:,:,k-1))/log(3);
433     end
434 end

```

Display the results, and end the function.

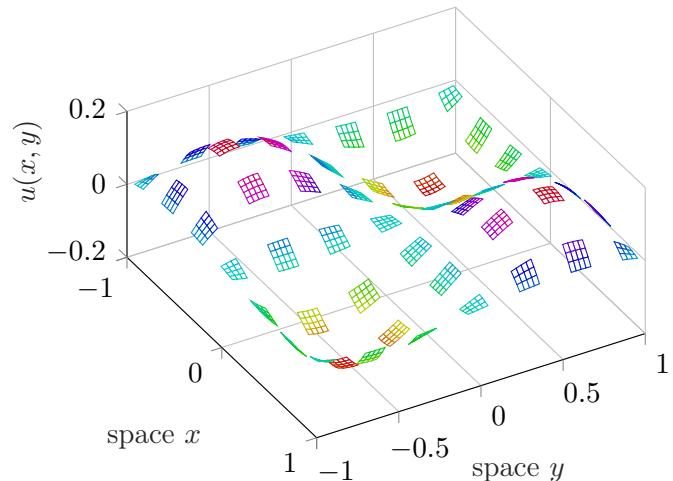
```

440 rmsError=rmsError
441 errorRatios=errorRatios
442 orderInH=orderInH
443 end%function

```

Fin.

Figure 4.5: Equilibrium of the macroscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.7). The patch size is not small so we can see the patches.



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

Here we find the steady state $u(x, y)$, see Figure 4.5, to the heterogeneous PDE (inspired by Freese et al.⁹ §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

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

and for forcing $f(x, y)$ such that the exact equilibrium is $u = x(1 - e^{1-|x|})y(1 - e^{1-|y|})$. 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.

⁹ <http://arxiv.org/abs/2211.13731>

```

70 nPatch = 7
71 nSubP = 5
72 dx = 0.03

Specify some order of interpolation.

78 configPatches2(@monoscaleDiffForce2,[-1 1 -1 1],'equispace' ...
79 ,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 4.7.1](#).

```

87 x=patches.x; y=patches.y;
88 patches.A = sin(pi*x).*sin(pi*y);
89 patches.fu = ...
90     +2*patches.A.*((9*x.^2.*y.^2-3*x.^2-3*y.^2+1) ...
91     +12*x.*y.*((x.^2+y.^2-2));

```

By construction, the PDE has analytic solution

```
97 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.

```

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

```

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

```

121 tic;
122 uSoln = fsolve(@theRes,u0(patches.i) ...
123     ,optimoptions('fsolve','Display','off'));
124 solnTime = toc
125 normResidual = norm(theRes(uSoln))
126 normSoln = norm(uSoln)
127 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.

```

135 u0(patches.i) = uSoln;
136 u0 = patchEdgeInt2(u0);

```

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

```

147 figure(1), clf, colormap(0.8* hsv)
148 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'));
158 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
159 if0urCf2tex(mfilename)%optionally save

```

4.7.1 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

```

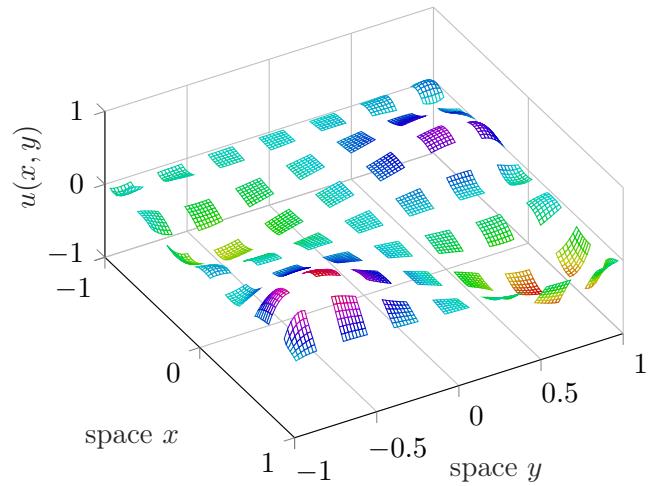
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one $,:$.

```

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 4.6: Equilibrium of the multiscale diffusion problem of Freese with Dirichlet zero-value boundary conditions (Section 4.8). 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.



4.8 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.¹⁰ §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.

```
41 clear all
42 global patches
43 %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.

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

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

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

¹⁰ <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’:

```
79 nPatch = 7
80 configPatches2(@twoscaleDiffForce2, [-1 1], 'equispace' ...
81 ,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 4.9.1](#).

```
89 x = patches.x; y = patches.y;
90 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.

```
104 u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
105 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;
106 patches.i = find(~isnan(u0));
107 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 3.17](#)), and give magnitudes.

```
116 tic;
117 uSoln = fsolve(@theRes,u0(patches.i) ...
118 ,optimoptions('fsolve','Display','off'));
119 solveTime = toc
120 normResidual = norm(theRes(uSoln))
121 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.

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

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

```
141 figure(1), clf, colormap(0.8* hsv)
142 x(end+1,:,:)=nan; u0(end+1,:,:)=nan;
143 y(:,end+1,:)=nan; u0(:,end+1,:)=nan;
144 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.

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

4.8.1 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 .

```

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

```

Set Dirichlet boundary value of zero around the square domain.

```

184     u( 1 ,:,:, :, 1 ,:) = 0; % left edge of left patches
185     u(end,:,:,:,end,:) = 0; % right edge of right patches
186     u(:, 1 ,:,:, :, 1 ) = 0; % bottom edge of bottom patches
187     u(:,end,:,:, :,end) = 0; % top edge of top patches

```

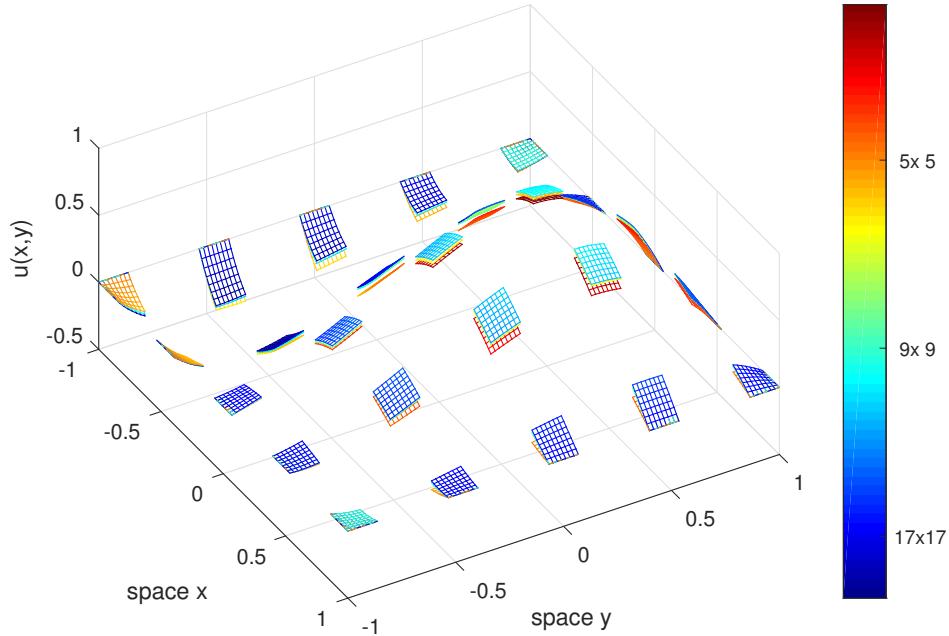
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one $,:$.

```

195     ut(i,j,:) ...
196     = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(:,j),2,2)/dy^2 ...
197       +2*patches.cs(i,j).*( u(i+1,j+1,:)-u(i-1,j+1,:)) ...
198         -(u(i+1,j-1,:)+u(i-1,j-1,:))/(4*dx*dy) ...
199       -patches.fu(i,j,:);
200 end%function twoscaleDiffForce2

```

Figure 4.7: 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 4.9). We only compare solutions only in these 25 common patches.



4.9 twoscaleDiffEquil2Errs: errors in equilibria of a 2D two-scale heterogeneous diffusion via small patches

Here we find the steady state $u(x, y)$ to the heterogeneous PDE (inspired by Freese et al.¹¹ §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

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

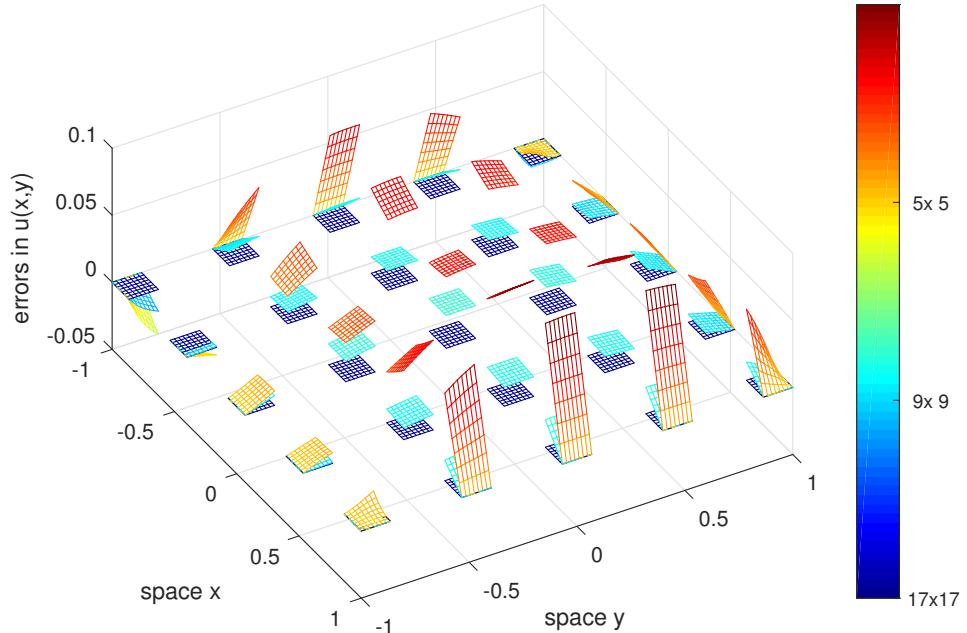
and for forcing $f := 10(x + y + \cos \pi x)$ (for which the solution has magnitude up to one).¹²

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

¹¹ <http://arxiv.org/abs/2211.13731>

¹² 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.¹³ For the same reason we do not invoke their smaller $\epsilon \approx 0.01$.

Figure 4.8: 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 4.9). 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).

```

81 clear all
82 global patches
83 %global OurCf2eps, OurCf2eps=true %option to save plot
84 switch 1
85     case 1, Dom.type = 'equispace'
86     case 2, Dom.type = 'chebyshev'
87     case 3, Dom.type = 'usergiven'
88 end% switch
89 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.

```

100 mPeriod = 6
101 z = (0.5:mPeriod)'/mPeriod;
102 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:

```

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

```

Set the periodicity ϵ , and other microscale parameters.

```

120 nPeriodsPatch = 1 % any integer
121 nSubP = nPeriodsPatch*mPeriod+2 % for edgy int
122 epsilon = 2/(nPatchMax*nPeriodsPatch+1/mPeriod)
123 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.

```

136 us=[]; xs=[]; ys=[]; nPs=[];
137 for log2N=log2Nmax:-1:2
138     if log2N==log2Nmax
139         Ps=linspace(1,nPatchMax ...
140             ,5-2*all(Dom.type=='chebyshev'))
141     else Ps=(Ps+1)/2
142 end

```

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

```
148 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).

```

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

```

Configure the patches:

```

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

```

Compute the time-constant forcing, and store in struct `patches` for access by the microcode of [Section 4.9.1](#).

```

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 3.17](#)).

```

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 = [nP;nP];

```

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;

```

```

309 end, hold off
310 view(60,55)
311 colorbar('Ticks',1:size(u,3) ...
312     , 'TickLabels', [num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
313 xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
314 if0urCf2eps([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.

```

326 err=u(:,:,:,1)-u;
327 maxAbsErr=max(abs(err(:)));
328 figure(2), clf, colormap(jet)
329 for p=1:size(u,3)
330     mesh(x(:),y(:),err(:,:,p)',p+err(:,:,p)'/maxAbsErr);
331     hold on;
332 end, hold off
333 view(60,55)
334 colorbar('Ticks',1:size(u,3) ...
335     , 'TickLabels', [num2str(nPs) ['x';'x';'x'] num2str(nPs)]);
336 xlabel('space x'), ylabel('space y')
337 zlabel('errors in u(x,y)')
338 if0urCf2eps(mfilename)%optionally save

```

4.9.1 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 .

```

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

```

Set Dirichlet boundary value of zero around the square domain.

```

369 u( 1 ,:,:,:,:, 1 ,:) = 0; % left edge of left patches
370 u(end,:,:,:,end,:) = 0; % right edge of right patches
371 u(:, 1 ,:,:,:,:, 1 ) = 0; % bottom edge of bottom patches
372 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 $,:$

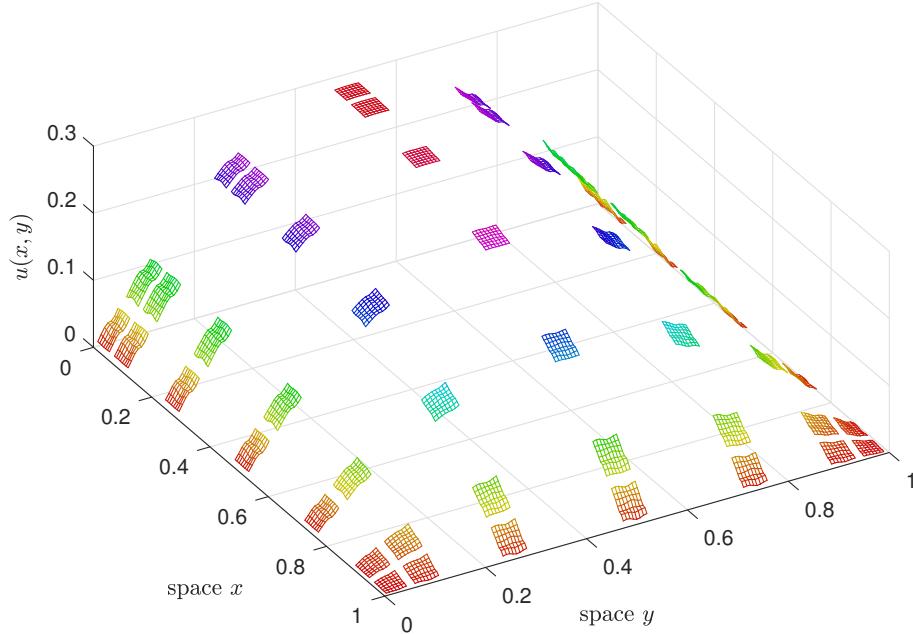
```

380 ut(i,j,:) ...
381 = 2*diff(u(:,j,:),2,1)/dx^2 +2*diff(u(i,:,:),2,2)/dy^2 ...
382     +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 4.9: Equilibrium of the macroscale diffusion problem of Abdulle with boundary conditions of Dirichlet zero-value except for $x = 0$ which is Neumann (Section 4.10). Here the patches have a Chebyshev-like spatial distribution. The patch size is chosen large enough to see within.



4.10 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 4.9](#) 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.)

```

53 mPeriod = 6
54 x = (0.5:mPeriod)'/mPeriod; y=x';
55 a = (2+1.8*sin(2*pi*x))./(2+1.8*sin(2*pi*y)) ...
56     +(2+ sin(2*pi*y))./(2+1.8*sin(2*pi*x));
57 diffusivityRange = [min(a(:)) max(a(:))]
```

Set the periodicity ϵ , here big enough so we can see the patches, and other microscale parameters.

```

64 epsilon = 0.04
65 dx = epsilon/mPeriod
66 nPeriodsPatch = 1 % any integer
67 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 4.10.1](#)

```

78 Dom.bcOffset = zeros(2);
79 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’:

```

86 nPatch = 7
87 Dom.type='chebyshev';
88 configPatches2(@abdulleDiffForce2,[0 1],Dom ...
89 ,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.

```

102 u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
103 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;
104 patches.i = find(~isnan(u0));
105 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 3.17](#)), and give magnitudes.

```

114 tic;
115 uSoln = fsolve(@theRes,u0(patches.i) ...
116 ,optimoptions('fsolve','Display','off'));
117 solnTime = toc
118 normResidual = norm(theRes(uSoln))
119 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.

```

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

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

```

139 figure(1), clf, colormap(0.8*hsv)
140 patches.x(end+1,:,:)=nan; u0(end+1,:,:)=nan;
141 patches.y(:,end+1,:)=nan; u0(:,end+1,:)=nan;
142 u = reshape(permute(squeeze(u0),[1 3 2 4]) ...
143     , [numel(patches.x) numel(patches.y)]);

```

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

```

150 mesh(patches.x(:),patches.y(:,u')); view(60,55)
151 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
152 if0urCf2eps(mfilename) %optionally save plot

```

4.10.1 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 .

```

171 function ut = abdulleDiffForce2(t,u,patches)
172     dx = diff(patches.x(2:3)); % x space step
173     dy = diff(patches.y(2:3)); % y space step
174     i = 2:size(u,1)-1; % x interior points in a patch
175     j = 2:size(u,2)-1; % y interior points in a patch
176     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.

```

184     u( 1 ,:,:, :, 1 ,:) = 0; % left edge of left patches
185     u(end,:,:,:,end,:) = 0; % right edge of right patches
186     u(:, 1 ,:,:, :, 1 ) = 0; % bottom edge of bottom patches
187     u(:,end,:,:,:,end) = 0; % top edge of top patches
188     if 1, u(1,:,:,:,1,:) = u(2,:,:,:,1,:); end% left Neumann

```

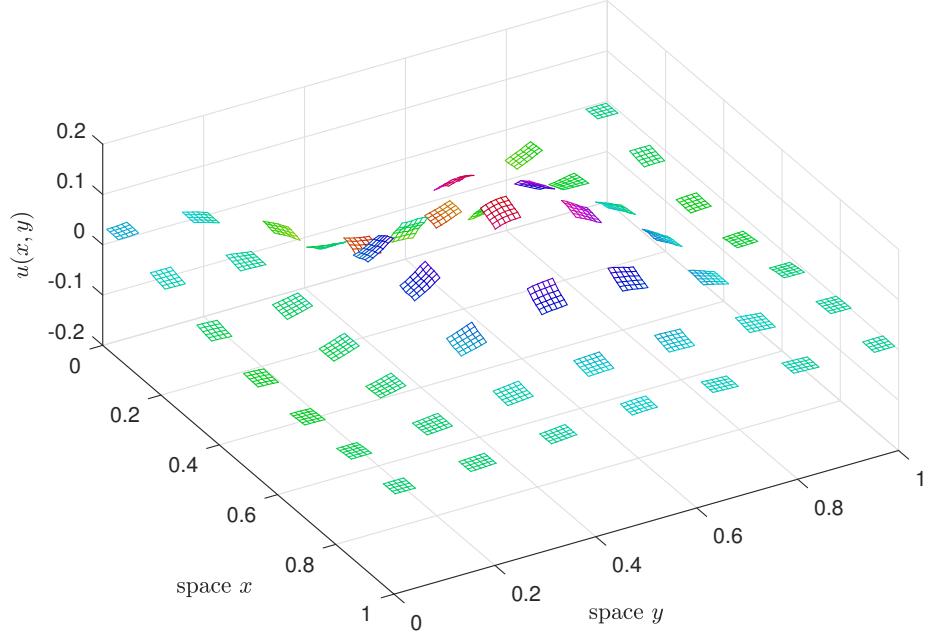
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one $,:$.

```

196     ut(i,j,:) = diff(patches.cs(:,j).*diff(u(:,j,:)))/dx^2 ...
197         + diff(patches.cs(i,:).*diff(u(i,:,:),1,2),1,2)/dy^2 ...
198         + 10;
199 end%function abdulleDiffForce2

```

Figure 4.10: Equilibrium of the macroscale diffusion problem of Bonizzoni et al. with Neumann boundary conditions of zero (Section 4.11). Here the patches have a equispaced spatial distribution. The microscale periodicity, and hence the patch size, is chosen large enough to see within.



4.11 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.¹⁴ §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, $\mu_1(x, y) \in [0.01, 0.1]$ ¹⁵ 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 4.10 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.

```
50 clear all
51 global patches
52 %global OurCf2eps, OurCf2eps=true %option to save plot
```

¹⁴ <http://arxiv.org/abs/2211.15221>

¹⁵ 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.

```
63 mPeriod = 4
64 mu1 = 0.01*10.^rand(mPeriod)
65 mu2 = 2*pi*rand(mPeriod)
66 cs = cat(3,mu1,cos(mu2),sin(mu2));
67 meanDiffAdvec=squeeze(mean(mean(cs)))
```

Set the periodicity ϵ , here big enough so we can see the patches, and other microscale parameters.

```
74 epsilon = 0.04
75 dx = epsilon/mPeriod
76 nPeriodsPatch = 1 % any integer
77 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:

```
89 nPatch = 7
90 Dom.type= 'equispace';
91 Dom.bcOffset = 0.5;
92 configPatches2(@randAdvecDiffForce2,[0 1],Dom ...
93 ,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 4.11.1](#).

```
101 mu = [ 0.25+0.5*rand(1,2) 0.1+0.15*rand ]
102 patches.fu = exp(-((patches.x-mu(1)).^2 ...
103 +(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.

```
118 u0 = zeros(nSubP,nSubP,1,1,nPatch,nPatch);
119 u0([1 end],:,:) = nan; u0(:,[1 end],:) = nan;
120 patches.i = find(~isnan(u0));
121 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 3.17](#)).

```
130 tic;
131 uSoln = fsolve(@theRes,u0(patches.i) ...
132 ,optimoptions('fsolve','Display','off'));
133 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

```

4.11.1 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 $,:$.

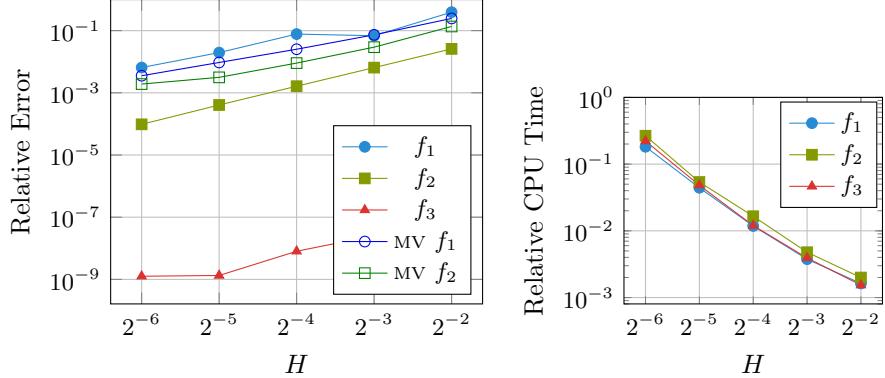
```

211     ut(i,j,:)
212     = patches.cs(i,j,1).*(diff(u(:,j,:),2,1)/dx^2 ...
213         +diff(u(:, :,1),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 4.11: results for the computational homogenisation of a forced, non-autonomous, 2D wave (Section 4.12). (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.



4.12 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 Section 4.5. It favourably compares to the examples of Maier & Verfürth (2021).

Figure 4.11 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 than 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.

```

68 clear all
69 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.

```

77 epsilon = 4*dx
78 nPeriodsPatch = 1
79 mPeriod = round(epsilon/dx)
80 nSubP = mPeriod*nPeriodsPatch+2

```

Choose which of three forcing functions to use

```
86 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.

```

99 nPatch = 2/epsilon/nPeriodsPatch
100 for iPat=0:9
101 if iPat>0, nPatch=nPatch/2, end
102 if nPatch<8, break, end

```

Set the periodic heterogeneous coefficient, isotropic:

$$a_\epsilon(t, x) = [3 + \sin(2\pi x/\epsilon) + \sin(2\pi t)] \cdot [3 + \sin(2\pi y/\epsilon) + \sin(2\pi t)],$$

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.

```

130 ratio = (nSubP-2)*dx/(2/nPatch)
131 Xleft=(1-ratio)/nPatch;
132 xmid=Xleft+dx*(0:mPeriod-1)'; % half-points
133 xi = Xleft+dx*(-0.5:mPeriod-1)'; % grid-points
134 % two components for ax, the x-dirn interactions
135 cHetr(:,:,1) = (3+cos(2*pi*xmid/epsilon))+0*xi';
136 cHetr(:,:,2) = 0*xmid+(3+cos(2*pi*xi'/epsilon));
137 % two components for ay, the y-dirn interactions
138 cHetr(:,:,3) = (3+cos(2*pi*xi/epsilon))+0*xmid';
139 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 .

```
148 configPatches2(@heteroWave2, [-1 1 -1 1], nan, nPatch ...
149     , 0, ratio, nSubP, 'EdgyInt', true, 'hetCoeffs', cHetr );
```

A check on the spatial geometry.

```
155 global patches
156 dxPat=diff(patches.x(1:2));
157 assert(abs(dx-dxPat)<1e-9, "dx mismatch")
```

Simulate Set the particular forcing function to use, and the zero initial conditions of a simulation.

```
167 patches.eff=fn;
168 clear uv0
169 uv0(:, :, 1, 1, :, :) = 0*patches.x+0*patches.y;
170 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`.

```
182 disp('Now simulate over time')
183 tic
184 [ts,us] = ode23(@patchSys2, linspace(0,1,11), uv0(:));
185 if iPAt==0, odeTime0=toc
186 else relodeTime(iPat)=toc/odeTime0
187 end
```

Compute error compared to full domain simulation Get spatial coordinates of patch-interior points, and reshape to column vectors.

```
197 i = 2:nSubP-1;
198 x = squeeze(patches.x(i,:,:,:, :, :));
199 y = squeeze(patches.y(:,i,:,:,:, :));
200 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.

```
208 uv = squeeze(patchEdgeInt2(us(end,:)));
209 u = squeeze(uv(i,i,1,:,:));
210 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.

```
217 if iPAt==0
218 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

```

4.12.1 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 $y = \pm 1$.

```
47 x=patches.x; y=patches.y;
48 l=find(abs(x)>1); x(l)=x(l)-sign(x(l))*2;
49 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.

```
59 f1 = ( (abs(x)>0.4)*(20*t+230*t^2) ...
60     +(abs(x)<0.4)*(100*t+2300*t^2) ).*sign(x).*sign(y);
61 f2 = 20*t*x.* (1-abs(x)).*y.* (1-abs(y)) ...
62     +230*t^2*(sign(y).*x.* (1-abs(x))+sign(x).*y.* (1-abs(y)));
63 f3 = (5*t+50*t^2)*sin(pi*x).*sin(pi*y);
```

Also set the heterogeneous interactions at this time.

```
69 ax = (patches.cs(:,:,1)+sin(2*pi*t)) ...
70     .* (patches.cs(:,:,2)+sin(2*pi*t));
71 ay = (patches.cs(:,:,3)+sin(2*pi*t)) ...
72     .* (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.

```
81 ut = nan+u; % preallocate output array
82 ut(i,j,1,:) = u(i,j,2,:);
83 ut(i,j,2,:) ...
84 = diff(ax(:,j).*diff(u(:,j,1,:),1),1)/dx^2 ...
85     +diff/ay(i,:).*diff(u(i,:,1,:),1,2),1,2)/dy^2 ...
86     +(patches.eff==1)*f1(i,j,:,:)
87     +(patches.eff==2)*f2(i,j,:,:)
88     +(patches.eff==3)*f3(i,j,:,:)
89     + 1e-4*(diff(u(:,j,2,:),2,1)/dx^2+diff(u(i,:,2,:),2,2)/dy^2);
90 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.

5 Patches in 3D space

5.1 configPatches3(): configures spatial patches in 3D

Section contents

5.1.1	If no arguments, then execute an example	205
5.1.2	heteroWave3(): heterogeneous Waves	208
5.1.3	Parse input arguments and defaults	208
5.1.4	The code to make patches	210
5.1.5	Set ensemble inter-patch communication	213

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)
```

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, . . . ; 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. So either $\text{ratio} = \frac{1}{2}$ 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 must be odd so that there is/are centre-patch micro-grid point/planes in each patch.

- ‘`nEdge`’ (not yet implemented), *optional*, default=1, for each patch, the number of face values set by interpolation at the face 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/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.
 - 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

```
218 if nargout==0, global patches, end
```

- `.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 × 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) × 1 × 1 × 1 × 1 × nPatch(1) × 1 × 1` array of the regular spatial locations x_{iI} of the microscale grid points in every patch.
- `.y` (8D) is `1 × nSubP(2) × 1 × 1 × 1 × 1 × nPatch(2) × 1` array of the regular spatial locations y_{jJ} of the microscale grid points in every patch.
- `.z` (8D) is `1 × 1 × nSubP(3) × 1 × 1 × 1 × 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` is, for each patch, the number of face values set by interpolation at the face regions of each patch.
- `.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

```
307 if nargin==0
308 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 (" \leftrightarrow " denotes function recursion).

1. configPatches3
2. ode23 integrator \leftrightarrow patchSys3 \leftrightarrow 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.

```
326 mPeriod = [2 2 2];
327 cHetr = exp(0.9*randn([mPeriod 3]));
328 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.

```
340 global patches
341 patches = configPatches3(@heteroWave3, [-pi pi], 'periodic' ...
342 , 5, 0, 0.22, mPeriod+2, 'EdgyInt', true ...
343 , 'hetCoeffs', cHetr);
```

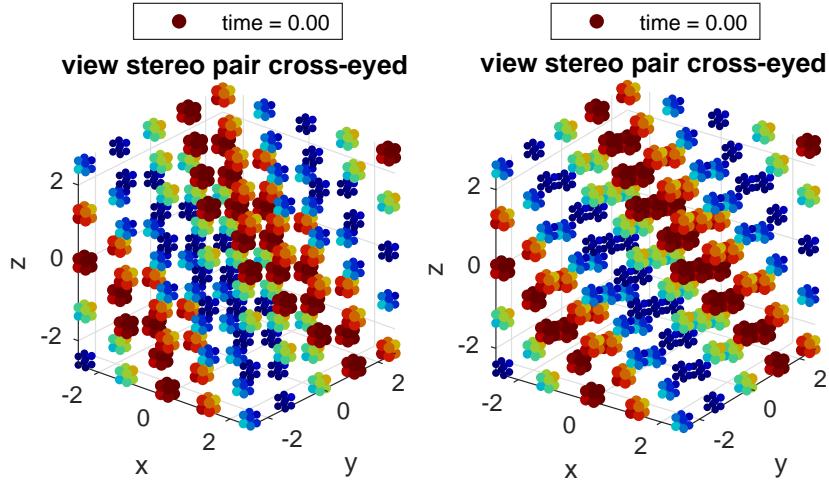
Set a wave initial state using auto-replication of the spatial grid, and as Figure 5.1 shows. This wave propagates diagonally across space. Concatenate the two u, v -fields to be the two components of the fourth dimension.

```
353 u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
354 v0 = -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
355 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).

```
372 disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
373 if ~exist('OCTAVE_VERSION', 'builtin')
374 [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
375 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 $t = 6$.



```

376     lsode_options('absolute tolerance',1e-4);
377     lsode_options('relative tolerance',1e-4);
378     [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
379 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.

```

393 figure(1), clf, colormap(0.8*jet)
394 xs = patches.x+0*patches.y+0*patches.z;
395 ys = patches.y+0*patches.x+0*patches.z;
396 zs = patches.z+0*patches.y+0*patches.x;
397 if 1, xs([1 end],:,:)=nan;
398     xs(:,[1 end],:,:)=nan;
399     xs(:,:,1 end,:)=nan;
400 end;%option
401 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.

```

409 pix = @(u) 15*abs(u)+7;
410 col = @(u) sign(u).*abs(u);

```

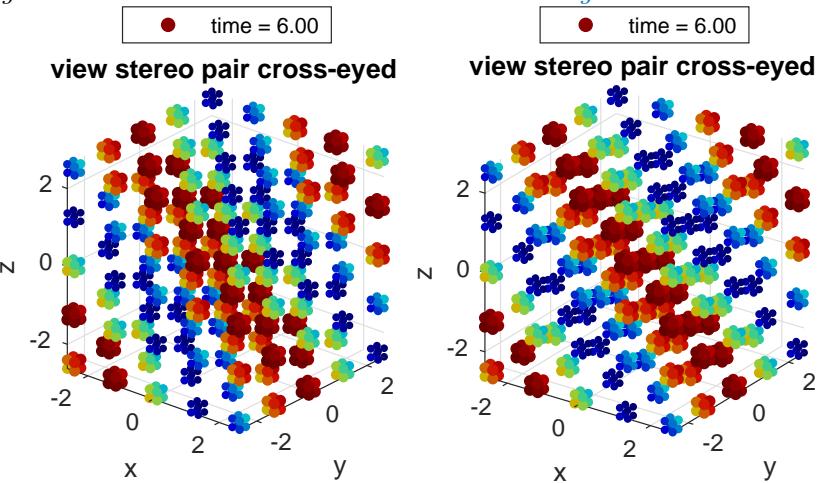
Loop to plot at each and every time step.

```

416 for i = 1:length(ts)
417     uv = patchEdgeInt3(us(i,:));
418     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.



```

419 for p=1:2
420 subplot(1,2,p)
421 if (i==1)| exist('OCTAVE_VERSION','builtin')
422 scat(p) = scatter3(xs(j),ys(j),zs(j),'filled');
423 axis equal, caxis(col([0 1])), view(45-5*p,25)
424 xlabel('x'), ylabel('y'), zlabel('z')
425 title('view stereo pair cross-eyed')
426 end % in matlab just update values
427 set(scat(p),'CData',col(u(j)) ...
428 , 'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
429 legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
430 end

```

Optionally save the initial condition to graphic file for Figure 4.1, and optionally save the last plot.

```

438 if i==1,
439 ifOurCf2eps([mfilename 'ic'])
440 disp('Type space character to animate simulation')
441 pause
442 else pause(0.05)
443 end
444 end% i-loop over all times
445 ifOurCf2eps([mfilename 'fin'])

```

Upon finishing execution of the example, exit this function.

```

460 return
461 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_{ijk}^x , c_{ijk}^y and c_{ijk}^z , 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
```

5.1.3 Parse input arguments and defaults

```
478 p = inputParser;
479 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
480 addRequired(p,'fun',fnValidation);
481 addRequired(p,'Xlim',@isnumeric);
482 %addRequired(p,'Dom'); % too flexible
483 addRequired(p,'nPatch',@isnumeric);
484 addRequired(p,'ordCC',@isnumeric);
485 addRequired(p,'dx',@isnumeric);
486 addRequired(p,'nSubP',@isnumeric);
487 addParameter(p,'nEdge',1,@isnumeric);
488 addParameter(p,'EdgyInt',false,@islogical);
489 addParameter(p,'nEnsem',1,@isnumeric);
490 addParameter(p,'hetCoeffs',[],@isnumeric);
```

```

491 addParameter(p,'parallel',false,@islogical);
492 %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
493 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

499 patches.nEdge = p.Results.nEdge;
500 patches.EdgyInt = p.Results.EdgyInt;
501 patches.nEnsem = p.Results.nEnsem;
502 cs = p.Results.hetCoeffs;
503 patches.parallel = p.Results.parallel;
504 %patches.nCore = p.Results.nCore;

```

Initially duplicate parameters for three space dimensions as needed.

```

512 if numel(Xlim)==2, Xlim = repmat(Xlim,1,3); end
513 if numel(nPatch)==1, nPatch = repmat(nPatch,1,3); end
514 if numel(dx)==1, dx = repmat(dx,1,3); end
515 if numel(nSubP)==1, nSubP = repmat(nSubP,1,3); end

```

Check parameters.

```

522 assert(Xlim(1)<Xlim(2) ...
523     , 'first pair of Xlim must be ordered increasing')
524 assert(Xlim(3)<Xlim(4) ...
525     , 'second pair of Xlim must be ordered increasing')
526 assert(Xlim(5)<Xlim(6) ...
527     , 'third pair of Xlim must be ordered increasing')
528 assert(patches.nEdge==1 ...
529     , 'multi-edge-value interp not yet implemented')
530 assert(all(2*patches.nEdge<nSubP) ...
531     , 'too many edge values requested')
532 %if patches.nCore>1
533 %    warning('nCore>1 not yet tested in this version')
534 %end

```

For compatibility with pre-2023 functions, if parameter Dom is Nan, then we set the ratio to be the value of the so-called dx vector.

```

545 if ~isstruct(Dom), pre2023=isnan(Dom);
546 else pre2023=false; end
547 if pre2023, ratio=dx; dx=nan; end

```

Default macroscale conditions are periodic with evenly spaced patches.

```

556 if isempty(Dom), Dom=struct('type','periodic'); end
557 if (~isstruct(Dom))&isnan(Dom), Dom=struct('type','periodic'); end

```

If Dom is a string, then just set type to that string, and subsequently set corresponding defaults for others fields.

```

565 if ischar(Dom), Dom=struct('type',Dom); end

```

We allow different macroscale domain conditions in the different directions. But for the moment do not allow periodic to be mixed with the others (as the

interpolation mechanism is different code)—hence why we choose `periodic` to be seven characters, whereas the others are eight characters. The different conditions are coded in different rows of `Dom.type`, so we duplicate the string if only one row specified.

```
578 if size(Dom.type,1)==1, Dom.type=repmat(Dom.type,3,1); end
```

Check what is and is not specified, and provide default of Dirichlet boundaries if no `bcOffset` specified when needed. Do so for all three directions independently.

```
586 patches.periodic=false;
587 for p=1:3
588 switch Dom.type(:, :)
589 case 'periodic'
590     patches.periodic=true;
591     if isfield(Dom,'bcOffset')
592         warning('bcOffset not available for Dom.type = periodic'), end
593         msg=' not available for Dom.type = periodic';
594         if isfield(Dom,'X'), warning(['X' msg]), end
595         if isfield(Dom,'Y'), warning(['Y' msg]), end
596         if isfield(Dom,'Z'), warning(['Z' msg]), end
597     case {'equispace','chebyshev'}
598         if ~isfield(Dom,'bcOffset'), Dom.bcOffset=zeros(2,3); end
599         % for mixed with usergiven, following should still work
600         if numel(Dom.bcOffset)==1
601             Dom.bcOffset=repmat(Dom.bcOffset,2,3); end
602         if numel(Dom.bcOffset)==3
603             Dom.bcOffset=repmat(Dom.bcOffset(:, ),2,1); end
604             msg=' not available for Dom.type = equispace or chebyshev';
605             if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
606             if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
607             if (p==3)& isfield(Dom,'Z'), warning(['Z' msg]), end
608     case 'usergiven'
609         % if isfield(Dom,'bcOffset')
610         % warning('bcOffset not available for usergiven Dom.type'), end
611         msg=' required for Dom.type = usergiven';
612         if p==1, assert(isfield(Dom,'X'),['X' msg]), end
613         if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
614         if p==3, assert(isfield(Dom,'Z'),['Z' msg]), end
615     otherwise
616         error([Dom.type ' is unknown Dom.type'])
617     end%switch Dom.type
618 end%for p
```

5.1.4 The code to make patches

First, store the pointer to the time derivative function in the struct.

```
632 patches.fun = fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 or (not yet??) -1.

```
641 assert((ordCC>=-1) & (floor(ordCC)==ordCC), ...
642     'ordCC out of allowed range integer>=-1')
```

For odd `ordCC` do interpolation based upon odd neighbouring patches as is useful for staggered grids.

```
649 patches.stag = mod(ordCC,2);
650 assert(patches.stag==0,'staggered not yet implemented??')
651 ordCC = ordCC+patches.stag;
652 patches.ordCC = ordCC;
```

Check for staggered grid and periodic case.

```
658 if patches.stag, assert(all(mod(nPatch,2)==0), ...
659     'Require an even number of patches for staggered grid')
660 end
```

Set the macro-distribution of patches Third, set the centre of the patches in the macroscale grid of patches. Loop over the coordinate directions, setting the distribution into `Q` and finally assigning to array of corresponding direction.

```
675 for q=1:3
676 qq=2*q-1;
```

Distribution depends upon `Dom.type`:

```
682 switch Dom.type(q,:)
```

The periodic case is evenly spaced within the spatial domain. Store the size ratio in `patches`.

```
690 case 'periodic'
691     Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
692     DQ=Q(2)-Q(1);
693     Q=Q(1:nPatch(q))+diff(Q)/2;
694     pEI=patches.EdgyInt;% abbreviation
695     if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-1-pEI)*(2-pEI);
696     else         ratio(q) = dx(q)/DQ*(nSubP(q)-1-pEI)/(2-pEI);
697     end
698     patches.ratio=ratio;
```

The equi-spaced case is also evenly spaced but with the extreme edges aligned with the spatial domain boundaries, modified by the offset.

```
707 case 'equispace'
708     Q=linspace(Xlim(qq)+((nSubP(q)-1)/2-Dom.bcOffset(qq))*dx(q) ...
709             ,Xlim(qq+1)-((nSubP(q)-1)/2-Dom.bcOffset(qq+1))*dx(q) ...
710             ,nPatch(q));
711     DQ=diff(Q(1:2));
```

```

712     width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
713     if DQ<width*0.999999
714         warning('too many equispace patches (double overlapping)')
715     end

```

The Chebyshev case is spaced according to the Chebyshev distribution in order to reduce macro-interpolation errors, $Q_i \propto -\cos(i\pi/N)$, but with the extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible ‘boundary layers’.²

```

732 case 'chebyshev'
733     halfWidth=dx(q)*(nSubP(q)-1)/2;
734     Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
735     Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
736 % Q = (Q1+Q2)/2-(Q2-Q1)/2*cos(linspace(0,pi,nPatch));

```

Search for total width of ‘boundary layers’ so that in the interior the patches are non-overlapping Chebyshev. But the width for assessing overlap of patches is the following variable `width`.

```

745     width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx(q);
746     for b=0:2:nPatch(q)-2
747         DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
748         if DQmin>width, break, end
749     end%for
750     if DQmin<width*0.999999
751         warning('too many Chebyshev patches (mid-domain overlap)')
752     end%if

```

Assign the centre-patch coordinates.

```

758     Q =[ Q1+(0:b/2-1)*width ...
759             (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...
760             Q2+(1-b/2:0)*width ];

```

The user-given case is entirely up to a user to specify, we just ensure it has the correct shape of a row.

```

769 case 'usergiven'
770     if q==1, Q = reshape(Dom.X,1,[]); end
771     if q==2, Q = reshape(Dom.Y,1,[]); end
772     if q==3, Q = reshape(Dom.Z,1,[]); end
773 end%switch Dom.type

```

Assign Q -coordinates to the correct spatial direction. At this stage they are all rows.

```

780 if q==1, X=Q; end
781 if q==2, Y=Q; end

```

² However, maybe overlapping patches near a boundary should be viewed as some sort of spatially analogue of the ‘christmas tree’ of projective integration and its integration to a slow manifold. Here maybe the overlapping patches allow for a ‘christmas tree’ approach to the boundary layers. Needs to be explored??

```

782 if q==3, Z=Q; end
783 end%for q

```

Construct the micro-grids Construct the microscale in each patch. Reshape the grid to be 8D to suit dimensions (micro,Vars,Ens,macro).

```

798 nSubP = reshape(nSubP,1,3); % force to be row vector
799 assert(patches.EdgyInt | all(mod(nSubP,2)==1), ...
800     'configPatches3: nSubP must be odd')
801 i0 = (nSubP(1)+1)/2;
802 patches.x = reshape( dx(1)*(-i0+1:i0-1)'+X ...
803     ,nSubP(1),1,1,1,1,nPatch(1),1,1);
804 i0 = (nSubP(2)+1)/2;
805 patches.y = reshape( dx(2)*(-i0+1:i0-1)'+Y ...
806     ,1,nSubP(2),1,1,1,1,nPatch(2),1);
807 i0 = (nSubP(3)+1)/2;
808 patches.z = reshape( dx(3)*(-i0+1:i0-1)'+Z ...
809     ,1,1,nSubP(3),1,1,1,1,nPatch(3));

```

Pre-compute weights for macro-periodic In the case of macro-periodicity, precompute the weightings to interpolate field values for coupling.³

```

821 if patches.periodic
822     ratio = reshape(ratio,1,3); % force to be row vector
823     patches.ratio = ratio;
824     if ordCC>0
825         [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
826         patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
827     end%if
828 end%if patches.periodic

```

5.1.5 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

- the right-face/centre realisations `1:nEnsem` are to interpolate to left-face `le`, and
- the left-face/centre realisations `1:nEnsem` are to interpolate to `re`.

`re` and `li` are ‘transposes’ of each other as `re(li)=le(ri)` are both `1:nEnsem`. Similarly for bottom-face/centre interpolation to top-face via `to`, top-face/centre interpolation to bottom-face via `bo`, back-face/centre interpolation to front-face via `fr`, and front-face/centre interpolation to back-face via `ba`.

The default is nothing shifty. This setting reduces the number of if-statements in function `patchEdgeInt3()`.

```

857 nE = patches.nEnsem;
858 patches.le = 1:nE; patches.ri = 1:nE;

```

³ **ToDo:** Might sometime extend to coupling via derivative values.

```

859 patches.bo = 1:nE; patches.to = 1:nE;
860 patches.ba = 1:nE; patches.fr = 1:nE;

```

However, if heterogeneous coefficients are supplied via `hetCoeffs`, then do some non-trivial replications. First, get microscale periods, patch size, and replicate many times in order to subsequently sub-sample: `nSubP` times should be enough. If `cs` is more than 4D, then the higher-dimensions are reshaped into the 4th dimension.

```

872 if ~isempty(cs)
873     [mx,my,mz,nc] = size(cs);
874     nx = nSubP(1); ny = nSubP(2); nz = nSubP(3);
875     cs = repmat(cs,nSubP);

```

If only one member of the ensemble is required, then sub-sample to patch size, and store coefficients in `patches` as is.

```

883 if nE==1, patches.cs = cs(1:nx-1,1:ny-1,1:nz-1,:); else

```

But for `nEnsem` > 1 an ensemble of $m_x m_y m_z$ phase-shifts of the coefficients is constructed from the over-supply. Here code phase-shifts over the periods—the phase shifts are like Hankel-matrices.

```

893 patches.nEnsem = mx*my*mz;
894 patches.cs = nan(nx-1,ny-1,nz-1,nc,mx,my,mz);
895 for k = 1:mz
896     ks = (k:k+nz-2);
897     for j = 1:my
898         js = (j:j+ny-2);
899         for i = 1:mx
900             is = (i:i+nx-2);
901             patches.cs(:,:, :, :, i, j, k) = cs(is,js,ks,:);
902         end
903     end
904 end
905 patches.cs = reshape(patches.cs,nx-1,ny-1,nz-1,nc,[]);

```

Further, set a cunning left/right/bottom/top/front/back realisation of inter-patch coupling. The aim is to preserve symmetry in the system when also invoking `EdgyInt`. What this coupling does without `EdgyInt` is unknown. Use auto-replication.

```

915 mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
916 le = mod(mmx+mod(nx-2, mx), mx)+1;
917 patches.le = reshape( le+mx*(mmy+my*mmz) ,[],1);
918 ri = mod(mmx-mod(nx-2, mx), mx)+1;
919 patches.ri = reshape( ri+mx*(mmy+my*mmz) ,[],1);
920 bo = mod(mmy+mod(ny-2, my), my)+1;
921 patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) ,[],1);
922 to = mod(mmy-mod(ny-2, my), my)+1;
923 patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) ,[],1);
924 ba = mod(mmz+mod(nz-2, mz), mz)+1;
925 patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) ,[],1);

```

```

926     fr = mod(mmxz-mod(nz-2,mz),mz)+1;
927     patches.fr = reshape( 1+mmx+mx*(mmy+my*(fr-1)) ,[],1);
Issue warning if the ensemble is likely to be affected by lack of scale separation.  

4
935 if prod(ratio)*patches.nEnsem>0.9, warning( ...
936 'Probably poor scale separation in ensemble of coupled phase-shifts')
937 scaleSeparationParameter = ratio*patches.nEnsem
938 end
End the two if-statements.
944 end%if-else nEnsem>1
945 end%if not-empty(cs)

```

If parallel code then first assume this is not within an `spmd`-environment, and so we invoke `spmd...end` (which starts a parallel pool if not already started). At this point, the global `patches` is copied for each worker processor and so it becomes *composite* when we distribute any one of the fields. Hereafter, *all fields in the global variable patches must only be referenced within an spmd-environment.*⁵

```

964 if patches.parallel
965     spmd

```

Second, decide which dimension is to be sliced among parallel workers (for the moment, do not consider slicing the ensemble). Choose the direction of most patches, biased towards the last.

```

974 [~,pari]=max(nPatch+0.01*(1:3));
975 patches.codist=codistributor1d(5+pari);

```

`patches.codist.Dimension` is the index that is split among workers. Then distribute the appropriate coordinate direction among the workers: the function must be invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```

985 switch pari
986     case 1, patches.x=codistributed(patches.x,patches.codist);
987     case 2, patches.y=codistributed(patches.y,patches.codist);
988     case 3, patches.z=codistributed(patches.z,patches.codist);
989 otherwise
990     error('should never have bad index for parallel distribution')
991 end%switch
992 end%spmd

```

If not parallel, then clean out `patches.codist` if it exists. May not need, but safer.

⁴ **ToDo:** Need to justify this and the arbitrary threshold more carefully??

⁵ If subsequently outside `spmd`, then one must use functions like `getfield(patches{1}, 'a')`.

```
1000 else% not parallel
1001     if isfield(patches,'codist'), rmfield(patches,'codist'); end
1002 end%if-parallel
```

Fin

```
1011 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. It assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables ([Section 5.1](#)) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```
28 function dudt = patchSys3(t,u,patches,varargin)
29 if nargin<3, global patches, end
```

Input

- `u` is a vector/array of length $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ where there are `nVars` · `nEnsem` field values at each of the points in the $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nSubP}(3) \times \text{nPatch}(1) \times \text{nPatch}(2) \times \text{nPatch}(3)$ 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 $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nSubP}(3) \times \text{nVars} \times \text{nEnsem} \times \text{nPatch}(1) \times \text{nPatch}(2) \times \text{nPatch}(3)$. Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
 - `.x` is $\text{nSubP}(1) \times 1 \times 1 \times 1 \times \text{nPatch}(1) \times 1 \times 1$ 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 $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times 1 \times \text{nPatch}(2) \times 1$ 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 $1 \times 1 \times \text{nSubP}(3) \times 1 \times 1 \times 1 \times \text{nPatch}(3)$ 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 number of parameters to be passed onto the users time-derivative function as specified in `configPatches3`.

Output

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

Sets the edge-face values from macroscale interpolation of centre-patch values, and if necessary, reshapes the fields `u` as a 8D-array. [Section 5.3](#) describes `patchEdgeInt3()`.

```
109 sizeu = size(u);
110 u = patchEdgeInt3(u, patches);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge/face values with the dummy value of zero (as `ode15s` chokes on NaNs), then return to the user/integrator as same sized array as input.

```
121 dudt = patches.fun(t,u,patches,varargin{:});
122 dudt([1 end],:,:,:,:, :, :) = 0;
123 dudt(:,[1 end],:,:,:,:, :, :) = 0;
124 dudt(:,:, [1 end],:,:,:,:, :, :) = 0;
125 dudt = reshape(dudt, sizeu);
```

Fin.

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.⁶

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 $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$ 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 $\text{nSubP1} \times 1 \times 1 \times 1 \times 1 \times \text{nPatch1} \times 1 \times 1$ 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 $1 \times \text{nSubP2} \times 1 \times 1 \times 1 \times 1 \times \text{nPatch2} \times 1$ 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 $1 \times 1 \times \text{nSubP3} \times 1 \times 1 \times 1 \times \text{nPatch3}$ 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 $\{0, 2, 4, \dots\}$
 - **.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.

⁶ Script **patchEdgeInt3test.m** verifies this code.

- `.Cwtsr` and `.Cwts1` 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).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

Output

- `u` is 8D array, $nSubP1 \cdot nSubP2 \cdot nSubP3 \cdot nVars \cdot nEnsem \cdot nPatch1 \cdot nPatch2 \cdot nPatch3$, of the fields with face values set by interpolation.

Test for reality of the field values, and define a function accordingly. Could be problematic if some variables are real and some are complex, or if variables are of quite different sizes.

```

124     if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
125         uclean=@(u) real(u);
126     else uclean=@(u) u;
127     end

```

Determine the sizes of things. Any error arising in the reshape indicates `u` has the wrong size.

```

135 [~,~,~,~,~,~,~,Nz] = size(patches.z);
136 [~,ny,~,~,~,~,Ny,~] = size(patches.y);
137 [nx,~,~,~,~,Nx,~,~] = size(patches.x);
138 nEnsem = patches.nEnsem;
139 nVars = round( numel(u)/numel(patches.x) ...
140                 /numel(patches.y)/numel(patches.z)/nEnsem );
141 assert(numel(u) == nx*ny*nz*Nx*Ny*Nz*nVars*nEnsem ...
142         , 'patchEdgeInt3: input u has wrong size for parameters')
143 u = reshape(u,[nx ny nz nVars nEnsem Nx Ny Nz]);

```

For the moment assume the physical domain is either macroscale periodic or macroscale rectangle so that the coupling formulas are simplest. These index vectors point to patches and, if periodic, their six immediate neighbours.

```

153 I=1:Nx; Ip=mod(I,Nx)+1; Im=mod(I-2,Nx)+1;
154 J=1:Ny; Jp=mod(J,Ny)+1; Jm=mod(J-2,Ny)+1;
155 K=1:Nz; Kp=mod(K,Nz)+1; Km=mod(K-2,Nz)+1;

```

The centre of each patch (as `nx`, `ny` and `nz` are odd for centre-patch interpolation) is at indices

```

163 i0 = round((nx+1)/2);
164 j0 = round((ny+1)/2);
165 k0 = round((nz+1)/2);

```

5.3.1 Periodic macroscale interpolation schemes

```
174 if patches.periodic
```

Get the size ratios of the patches in each direction.

```
180 rx = patches.ratio(1);
181 ry = patches.ratio(2);
182 rz = patches.ratio(3);
```

5.3.1.1 Lagrange interpolation gives patch-face values

Compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Here the domain is macro-periodic.

```
193 ordCC = patches.ordCC;
194 if ordCC>0 % then finite-width polynomial interpolation
```

Interpolate the three directions in succession, in this way we naturally fill-in face-edge and corner values. Start with x -direction, and give most documentation for that case as the others are essentially the same.

x -normal face values The patch-edge values are either interpolated from the next-to-edge-face values, or from the centre-cross-plane values (not the patch-centre value itself as that seems to have worse properties in general). Have not yet implemented core averages.

```
210 if patches.EdgyInt % interpolate next-to-face values
211   U = u([2 nx-1],2:(ny-1),2:(nz-1),:,:,I,J,K);
212 else % interpolate centre-cross values
213   U = u(i0,2:(ny-1),2:(nz-1),:,:,I,J,K);
214 end;%if patches.EdgyInt
```

Just in case the last array dimension(s) are one, we have to force a padding of the sizes, then adjoin the extra dimension for the subsequent array of differences.

```
222 szU0=size(U); szU0=[szU0 ones(1,8-length(szU0)) ordCC];
```

Use finite difference formulas for the interpolation, so store finite differences ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) in these arrays. When parallel, in order to preserve the distributed array structure we use an index at the end for the differences.

```
232 if ~patches.parallel, dmu = zeros(szU0); % 9D
233 else dmu = zeros(szU0,patches.codist); % 9D
234 end%if patches.parallel
```

First compute differences $\mu\delta$ and δ^2 .

```
240 if patches.stag % use only odd numbered neighbours
241   error('polynomial interpolation not yet for staggered patch coupling')
242 %   dmux(:,:,(:,:,I,:,:,:1) = (Ux(:,:,(:,:,I,:,:,1)) + Ux(:,:,(:,:,I,:,:,1))/2
243 %   dmux(:,:,(:,:,I,:,:,:2) = (Ux(:,:,(:,:,I,:,:,2)) - Ux(:,:,(:,:,I,:,:,2));
244 %   Ip = Ip(Ip); Im = Im(Im); % increase shifts to \pm2
```

```

245 % dmuy(:,:,(:,:,J,:,:) = (Ux(:,:,(:,:,J,:))+Ux(:,:,(:,:,Jm,:))/2;
246 % dmuy(:,:,(:,:,J,:,:) = (Ux(:,:,(:,:,J,:))-Ux(:,:,(:,:,Jm,:)); %
247 % Jp = Jp(Jp); Jm = Jm(Jm); % increase shifts to \pm2
248 % dmuz(:,:,(:,:,K,1) = (Ux(:,:,(:,:,K,:))+Ux(:,:,(:,:,Km)))/2;
249 % dmuz(:,:,(:,:,K,2) = (Ux(:,:,(:,:,K,:))-Ux(:,:,(:,:,Km)))); %
250 % Kp = Kp(Kp); Km = Km(Km); % increase shifts to \pm2
251 else %disp('starting standard interpolation')
252 dmu(:,:,(:,:,I,:,:,:1) = (U(:,:,(:,:,Ip,:,:)) ...
253 -U(:,:,(:,:,Im,:,:))/2; %\mu\delta
254 dmu(:,:,(:,:,I,:,:,:2) = (U(:,:,(:,:,Ip,:,:)) ...
255 -2*U(:,:,(:,:,I,:,:))+U(:,:,(:,:,Im,:,:))); %\delta^2
256 end% if stag

```

Recursively take δ^2 of these to form successively higher order centred differences in space.

```

263 for k = 3:ordCC
264     dmu(:,:,(:,:,I,:,:,:k) =      dmu(:,:,(:,:,Ip,:,:,:k-2) ...
265     -2*dmu(:,:,(:,:,I,:,:,:k-2)+dmu(:,:,(:,:,Im,:,:,:k-2));
266 end

```

Interpolate macro-values to be Dirichlet face values for each patch ([Roberts & Kevrekidis 2007](#), [Bunder et al. 2017](#)), using the weights pre-computed by `configPatches3()`. Here interpolate to specified order.

For the case where next-to-face values interpolate to the opposite face-values: when we have an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to`, `patches.bo`, `patches.fr` and `patches.ba`.

```

282 k=1+patches.EdgyInt; % use centre or two faces
283 u(nx,2:(ny-1),2:(nz-1),:,:,patches.ri,I,:,:,:) ...
284 = U(1,:,:,:,:,:)* (1-patches.stag) ...
285 +sum( shiftdim(patches.Cwtsr(:,1),-8).*dmu(1,:,:,:,:,:,:,:,:) ,9);
286 u(1 ,2:(ny-1),2:(nz-1),:,:,patches.le,I,:,:,:) ...
287 = U(k,:,:,:,:,:)* (1-patches.stag) ...
288 +sum( shiftdim(patches.Cwtsl(:,1),-8).*dmu(k,:,:,:,:,:,:,:,:) ,9);

```

y-normal face values Interpolate from either the next-to-edge-face values, or the centre-cross-plane values.

```

300 if patches.EdgyInt % interpolate next-to-face values
301     U = u(:,[2 ny-1],2:(nz-1),:,:,I,J,K);
302 else % interpolate centre-cross values
303     U = u(:,j0,2:(nz-1),:,:,I,J,K);
304 end;%if patches.EdgyInt

```

Adjoin extra dimension for the array of differences.

```
310 szU0=size(U); szU0=[szU0 ones(1,8-length(szU0)) ordCC];
```

Store finite differences ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) in this array.

```

317     if ~patches.parallel, dmu = zeros(szU0); % 9D
318     else    dmu = zeros(szU0,patches.codist); % 9D
319     end%if patches.parallel

```

First compute differences $\mu\delta$ and δ^2 .

```

325     if patches.stag % use only odd numbered neighbours
326         error('polynomial interpolation not yet for staggered patch coupling')
327     else %disp('starting standard interpolation')
328         dmu(:,:,(:,:,J,:)) = (U(:,:,(:,:,Jp,:)) ...
329                                -U(:,:,(:,:,Jm,:))/2; %\mu\delta
330         dmu(:,:,(:,:,J,:)) = (U(:,:,(:,:,Jp,:)) ...
331                                -2*U(:,:,(:,:,J,:)) +U(:,:,(:,:,Jm,:)); %\delta^2
332     end% if stag

```

Recursively take δ^2 .

```

338     for k = 3:ordCC
339         dmu(:,:,(:,:,J,:)) =      dmu(:,:,(:,:,Jp,:),k-2) ...
340         -2*dmu(:,:,(:,:,J,:),k-2) +dmu(:,:,(:,:,Jm,:),k-2);
341     end

```

Interpolate macro-values using the weights pre-computed by `configPatches3()`.
An ensemble of configurations may have cross-coupling.

```

349 k=1+patches.EdgyInt; % use centre or two faces
350 u(:,ny,2:(nz-1),:,patches.to,:,J,:) ...
351     = U(:,1,:,:,:,:,*)*(1-patches.stag) ...
352     +sum( shiftdim(patches.Cwtsr(:,2),-8).*dmu(:,1,:,:,:,:,:,,:) ,9);
353 u(:,1 ,2:(nz-1),:,patches.bo,:,J,:) ...
354     = U(:,k,:,:,:,:,:,*)*(1-patches.stag) ...
355     +sum( shiftdim(patches.Cwtsl(:,2),-8).*dmu(:,k,:,:,:,:,:,,:) ,9);

```

z-normal face values Interpolate from either the next-to-edge-face values, or the centre-cross-plane values.

```

366     if patches.EdgyInt % interpolate next-to-face values
367         U = u(:,:,2 nz-1],:,:,I,J,K);
368     else % interpolate centre-cross values
369         U = u(:,:,k0,:,:,:,I,J,K);
370     end;%if patches.EdgyInt

```

Adjoin extra dimension for the array of differences.

```
376 szU0=size(U); szU0=[szU0 ones(1,8-length(szU0)) ordCC];
```

Store finite differences ($\mu\delta, \delta^2, \mu\delta^3, \delta^4, \dots$) in this array.

```

383     if ~patches.parallel, dmu = zeros(szU0); % 9D
384     else    dmu = zeros(szU0,patches.codist); % 9D
385     end%if patches.parallel

```

First compute differences $\mu\delta$ and δ^2 .

```

391 if patches.stag % use only odd numbered neighbours
392 error('polynomial interpolation not yet for staggered patch coupling')
393 else %disp('starting standard interpolation')
394 dmu(:,:,,:,(:,:,K,1) = (U(:,:,(:,:,(:,:,Kp) ...
395 -U(:,:,(:,:,(:,:,Km))/2; \%mu\delta
396 dmu(:,:,(:,:,(:,:,K,2) = (U(:,:,(:,:,(:,:,Kp) ...
397 -2*U(:,:,(:,:,(:,:,K) +U(:,:,(:,:,(:,:,Km)); \%delta^2
398 end% if stag

```

Recursively take δ^2 .

```

404 for k = 3:ordCC
405 dmu(:,:,(:,:,(:,:,K,k) = dmu(:,:,(:,:,(:,:,Kp,k-2) ...
406 -2*dmu(:,:,(:,:,(:,:,K,k-2) +dmu(:,:,(:,:,(:,:,Km,k-2);
407 end

```

Interpolate macro-values using the weights pre-computed by `configPatches3()`. An ensemble of configurations may have cross-coupling.

```

415 k=1+patches.EdgyInt; % use centre or two faces
416 u(:,:,nz,:,:patches.fr,:,:,:K) ...
417 = U(:,:,1,:,:,:,:)*(1-patches.stag) ...
418 +sum( shiftdim(patches.Cwtsr(:,3),-8).*dmu(:,:,1,:,:,:,:,:,:) ,9);
419 u(:,:,1,:,:patches.ba,:,:,:K) ...
420 = U(:,:,k,:,:,:,:)*(1-patches.stag) ...
421 +sum( shiftdim(patches.Cwtsl(:,3),-8).*dmu(:,:,k,:,:,:,:,:,:) ,9);

```

5.3.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

```
431 else% patches.ordCC<=0, spectral interpolation
```

We interpolate in terms of the patch index, I say, not directly in space. As the macroscale fields are N -periodic in the patch index I , the macroscale Fourier transform writes the centre-patch values as $U_I = \sum_k C_k e^{ik2\pi I/N}$. Then the face-patch values $U_{I\pm r} = \sum_k C_k e^{ik2\pi N(I\pm r)} = \sum_k C'_k e^{ik2\pi I/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For N patches we resolve ‘wavenumbers’ $|k| < N/2$, so set row vector $\mathbf{ks} = k2\pi/N$ for ‘wavenumbers’ $k = (0, 1, \dots, k_{\max}, -k_{\max}, \dots, -1)$ for odd N , and $k = (0, 1, \dots, k_{\max}, \pm(k_{\max} + 1) - k_{\max}, \dots, -1)$ for even N .

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches3` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch faces are near the middle of the gaps and swapped.

```

454 if patches.stag % transform by doubling the number of fields
455 error('staggered grid not yet implemented??')
456 v=nan(size(u)); % currently to restore the shape of u
457 u=cat(3,u(:,:,1:2:nPatch,:),u(:,:,2:2:nPatch,:));
458 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
459 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
460 r=r/2; % ratio effectively halved

```

```

461     nPatch=nPatch/2; % halve the number of patches
462     nVars=nVars*2;   % double the number of fields
463 else % the values for standard spectral
464     stagShift = 0;
465     iV = 1:nVars;
466 end%if patches.stag

```

Interpolate the three directions in succession, in this way we naturally fill-in face-edge and corner values. Start with x -direction, and give most documentation for that case as the others are essentially the same. Need these indices of patch interior.

```
476 ix = 2:nx-1;    iy = 2:ny-1;    iz = 2:nz-1;
```

x -normal face values Now set wavenumbers into a vector at the correct dimension. In the case of even N these compute the $+$ -case for the highest wavenumber zig-zag mode, $k = (0, 1, \dots, k_{\max}, +k_{\max} + 1 - k_{\max}, \dots, -1)$.

```

489 kMax = floor((Nx-1)/2);
490 kr = shiftdim( rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax) ,-4);

```

Compute the Fourier transform of the patch values on the centre-planes for all the fields. Unless doing patch-edgy interpolation when FT the next-to-face values. If there are an even number of points, then if complex, treat as positive wavenumber, but if real, treat as cosine. When using an ensemble of configurations, different configurations might be coupled to each other, as specified by `patches.le`, `patches.ri`, `patches.to`, `patches.bo`, `patches.fr` and `patches.ba`.

```

504 if ~patches.EdgyInt
505     Cm = fft( u(i0,iy,iz,:,:,:,:,:) ,[],6);
506     Cp = Cm;
507 else
508     Cm = fft( u( 2,iy,iz ,:,patches.le,:,:,:) ,[],6);
509     Cp = fft( u(nx-1,iy,iz ,:,patches.ri,:,:,:) ,[],6);
510 end%if ~patches.EdgyInt

```

Now invert the Fourier transforms to complete interpolation. Enforce reality when appropriate.

```

517 u(nx,iy,iz,:,:,:,:,:) = uclean( ifft( ...
518     Cm.*exp(1i*(stagShift+kr)) ,[],6) );
519 u( 1,iy,iz,:,:,:,:,:) = uclean( ifft( ...
520     Cp.*exp(1i*(stagShift-kr)) ,[],6) );

```

y -normal face values Set wavenumbers into a vector.

```

530 kMax = floor((Ny-1)/2);
531 kr = shiftdim( ry*2*pi/Ny*(mod((0:Ny-1)+kMax,Ny)-kMax) ,-5);

```

Compute the Fourier transform of the patch values on the centre-planes for all the fields.

```

538 if ~patches.EdgyInt
539     Cm = fft( u(:,j0,iz,:,:,:, :, :) ,[],7);
540     Cp = Cm;
541 else
542     Cm = fft( u(:,2    ,iz ,:, patches.bo,:,:,:) ,[],7);
543     Cp = fft( u(:,ny-1,iz ,:, patches.to,:,:,:) ,[],7);
544 end%if ~patches.EdgyInt

```

Invert the Fourier transforms to complete interpolation.

```

550 u(:,ny,iz,:,:,:, :, :) = uclean( ifft( ...
551     Cm.*exp(1i*(stagShift+kr)) ,[],7) );
552 u(:, 1,iz,:,:,:, :, :) = uclean( ifft( ...
553     Cp.*exp(1i*(stagShift-kr)) ,[],7) );

```

z-normal face values Set wavenumbers into a vector.

```

563 kMax = floor((Nz-1)/2);
564 kr = shiftdim( rz*2*pi/Nz*(mod((0:Nz-1)+kMax,Nz)-kMax) ,-6);

```

Compute the Fourier transform of the patch values on the centre-planes for all the fields.

```

571 if ~patches.EdgyInt
572     Cm = fft( u(:,:,k0,:,:,:, :, :) ,[],8);
573     Cp = Cm;
574 else
575     Cm = fft( u(:,:,2    ,:, patches.ba,:,:,:) ,[],8);
576     Cp = fft( u(:,:,nz-1 ,:, patches.fr,:,:,:) ,[],8);
577 end%if ~patches.EdgyInt

```

Invert the Fourier transforms to complete interpolation.

```

583 u(:,:,nz,:,:,:, :, :) = uclean( ifft( ...
584     Cm.*exp(1i*(stagShift+kr)) ,[],8) );
585 u(:,:, 1,:,:,:, :, :) = uclean( ifft( ...
586     Cp.*exp(1i*(stagShift-kr)) ,[],8) );
592 end% if ordCC>0

```

5.3.2 Non-periodic macroscale interpolation

```

603 else% patches.periodic false
604 assert(~patches.stag, ...
605 'not yet implemented staggered grids for non-periodic')

```

Determine the order of interpolation **px**, **py** and **pz** (potentially different in the different directions!), and hence size of the (forward) divided difference tables in **F** (9D) for interpolating to left/right, top/bottom, and front/back faces. Because of the product-form of the patch grid, and because we are doing *only* either edgy interpolation or cross-patch interpolation (*not* just the centre patch value), the interpolations are all 1D interpolations.

```

619 if patches.ordCC<1
620     px = Nx-1; py = Ny-1; pz = Nz-1;
621 else px = min(patches.ordCC,Nx-1);
622     py = min(patches.ordCC,Ny-1);
623     pz = min(patches.ordCC,Nz-1);
624 end
625 % interior indices of faces (ix n/a)
626 ix=2:nx-1; iy=2:ny-1; iz=2:nz-1;

```

5.3.2.1 x -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble. For `EdgyInt`, the ‘reversal’ of the next-to-face values are because their values are to interpolate to the opposite face of each patch.⁷

```

639 F = nan(patches.EdgyInt+1,ny-2,nz-2,nVars,nEnsem,Nx,Ny,Nz,px+1);
640 if patches.EdgyInt % interpolate next-to-face values
641     F(:,:,(:,:,1,:,:,1)) = u([nx-1 2],iy,iz,:,:,:,:,:);
642     X = patches.x([nx-1 2],(:,:,(:,:,1,:,:,1));
643 else % interpolate mid-patch cross-patch values
644     F(:,:,(:,:,1,:,:,1)) = u(i0,iy,iz,:,:,:,:,:);
645     X = patches.x(i0,:,:,:,:,:,:,:);
646 end%if patches.EdgyInt

```

Form tables of divided differences Compute tables of (forward) divided differences (e.g., [Wikipedia 2022](#)) for every variable, and across ensemble, and in both directions, and for all three types of faces (left/right, top/bottom, and front/back). Recursively find all divided differences in the respective direction.

```

659 for q = 1:px
660     i = 1:Nx-q;
661     F(:,:,(:,:,i,:,:,q+1)) ...
662     = ( F(:,:,(:,:,i+1,:,:,q))-F(:,:,(:,:,i,:,:,q)) ...
663         ./ (X(:,:,(:,:,i+q,:,:) - X(:,:,(:,:,i,:,:)));
664 end

```

Interpolate with divided differences Now interpolate to find the face-values on left/right faces at `Xface` for every interior `Y,Z`.

```
673 Xface = patches.x([1 nx],(:,:,(:,:,1,:,:,1));
```

Code Horner’s recursive evaluation of the interpolation polynomials. Indices `i` are those of the left face of each interpolation stencil, because the table is of forward differences. This alternative: the case of order p_x , p_y and p_z interpolation across the domain, asymmetric near the boundaries of the rectangular domain.

```

684 i = max(1,min(1:Nx,Nx-ceil(px/2))-floor(px/2));
685 Uface = F(:,:,(:,:,i,:,:,px+1));

```

⁷ **ToDo:** Have no plans to implement core averaging as yet.

```

686     for q = px:-1:1
687         Uface = F(:,:,(:,:,i,:,:,:q) ...
688             +(Xface-X(:,:,(:,:,i+q-1,:,:)).*Uface;
689     end

```

Finally, insert face values into the array of field values, using the required ensemble shifts.

```

697 u(1 ,iy,iz,:,:patches.le,:,:,:) = Uface(1,:,:,:,:,:,:,:);
698 u(nx, iy, iz, :, patches.ri, :, :, :) = Uface(2,:,:,:,:,:,:,:);

```

5.3.2.2 y -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble.

```

708 F = nan(nx,patches.EdgyInt+1,nz-2,nVars,nEnsem,Nx,Ny,Nz,py+1);
709 if patches.EdgyInt % interpolate next-to-face values
710     F(:,:,(:,:,1,:,:,:,:,1) = u(:,:,ny-1 2],iz,:,:,:,:,:);
711     Y = patches.y(:,:,ny-1 2],:,:,:,:,:,:);
712 else % interpolate mid-patch cross-patch values
713     F(:,:,(:,:,1,:,:,:,:,1) = u(:,j0,iz,:,:,:,:,:);
714     Y = patches.y(:,j0,:,:,:,:,:,:,:);
715 end%if patches.EdgyInt

```

Form tables of divided differences.

```

721 for q = 1:py
722     j = 1:Ny-q;
723     F(:,:,(:,:,j,:,:,q+1) ...
724     = ( F(:,:,(:,:,j+1,:,:,q)-F(:,:,(:,:,j,:,:,q)) ...
725         ./ (Y(:,:,(:,:,j+q,:)-Y(:,:,(:,:,j,:)));
726 end

```

Interpolate to find the top/bottom faces **Yface** for every x and interior z .

```
733 Yface = patches.y(:,[1 ny],:,:,:,:,:,:);
```

Code Horner’s recursive evaluation of the interpolation polynomials. Indices j are those of the bottom face of each interpolation stencil, because the table is of forward differences.

```

742 j = max(1,min(1:Ny,Ny-ceil(py/2))-floor(py/2));
743 Uface = F(:,:,(:,:,j,:,:,py+1);
744 for q = py:-1:1
745     Uface = F(:,:,(:,:,j,:,:,q) ...
746         +(Yface-Y(:,:,(:,:,j+q-1,:)).*Uface;
747 end

```

Finally, insert face values into the array of field values, using the required ensemble shifts.

```

755 u(:,1 ,iz,:,:patches.bo,:,:,:) = Uface(:,1,:,:,:,:,:,:,:);
756 u(:,ny,iz,:,:patches.to,:,:,:) = Uface(:,2,:,:,:,:,:,:,:);

```

5.3.2.3 z -direction values

Set function values in first ‘column’ of the tables for every variable and across ensemble.

```

766     F = nan(nx,ny,patches.EdgyInt+1,nVars,nEnsem,Nx,Ny,Nz,pz+1);
767     if patches.EdgyInt % interpolate next-to-face values
768         F(:,:,(:,:,(:,:,1)) = u(:,:,,[nz-1 2],(:,:,(:,:,1));
769         Z = patches.z(:,:,,[nz-1 2],(:,:,(:,:,1));
770     else % interpolate mid-patch cross-patch values
771         F(:,:,(:,:,(:,:,1)) = u(:,:,k0,:,:,:,(:,:,1));
772         Z = patches.z(:,:,k0,:,:,:,(:,:,1));
773     end%if patches.EdgyInt

```

Form tables of divided differences.

```

779     for q = 1:pz
780         k = 1:Nz-q;
781         F(:,:,(:,:,(:,:,k,q+1)) ...
782             = ( F(:,:,(:,:,(:,:,k+1,q))-F(:,:,(:,:,(:,:,k,q))) ...
783                 ./ (Z(:,:,(:,:,(:,:,k+q))-Z(:,:,(:,:,(:,:,k)));
784     end

```

Interpolate to find the face-values on front/back faces `Zface` for every x, y .

```
791     Zface = patches.z(:,:,,[1 nz],(:,:,(:,:,1));
```

Code Horner’s recursive evaluation of the interpolation polynomials. Indices k are those of the bottom face of each interpolation stencil, because the table is of forward differences.

```

800     k = max(1,min(1:Nz,Nz-ceil(pz/2))-floor(pz/2));
801     Uface = F(:,:,(:,:,(:,:,k,pz+1));
802     for q = pz:-1:1
803         Uface = F(:,:,(:,:,(:,:,k,q)) ...
804             +(Zface-Z(:,:,(:,:,(:,:,k+q-1))).*Uface;
805     end

```

Finally, insert face values into the array of field values, using the required ensemble shifts.

```

813     u(:,:,1 ,:,patches.fr,:,:,:) = Uface(:,:,1,:,:,:,:,:);
814     u(:,:,nz,:,:patches.ba,:,:,:) = Uface(:,:,2,:,:,:,:,:);

```

5.3.2.4 Optional NaNs for safety

We want a user to set outer face values on the extreme patches according to the microscale boundary conditions that hold at the extremes of the domain. Consequently, override their computed interpolation values with `NaN`.

```

826     u( 1,:,:,:, :, 1,:,:,:) = nan;
827     u(nx,:,:,:, :,Nx,:,:) = nan;
828     u(:, 1,:,:,:, :, 1,:) = nan;
829     u(:,ny,:,:,:, :,Ny,:) = nan;

```

```
830 u(:,:,1,:,:,:,1) = nan;
831 u(:,:,nz,:,:,:,1,Nz) = nan;
    End of the non-periodic interpolation code.
838 end%if patches.periodic else
    Fin, returning the 8D array of field values with interpolated faces.
849 end% function patchEdgeInt3
```

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 [Section 4.5](#). 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.

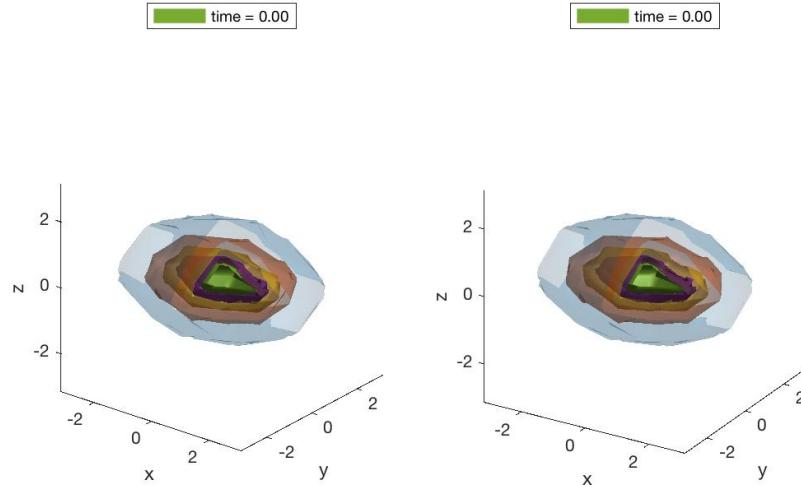
```
76 if ~exist('OCTAVE_VERSION','builtin')
77     [ts,us] = ode23(@patchSys3, 0.3*linspace(0,1,50).^2, u0(:));
78 else % octave version
79     [ts,us] = odeOcts(@patchSys3, 0.3*linspace(0,1).^2, u0(:));
80 end
```

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 $u = 0.1, 0.3, \dots, 0.9$, 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
163
164 end%for over time
165 if makeJpeg, print(['Figs/' mfilename 'tFin'], '-djpeg'), end

```

Finish the animation loop, and optionally output the isosurfaces of the final field, [Figure 5.4](#).

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

```

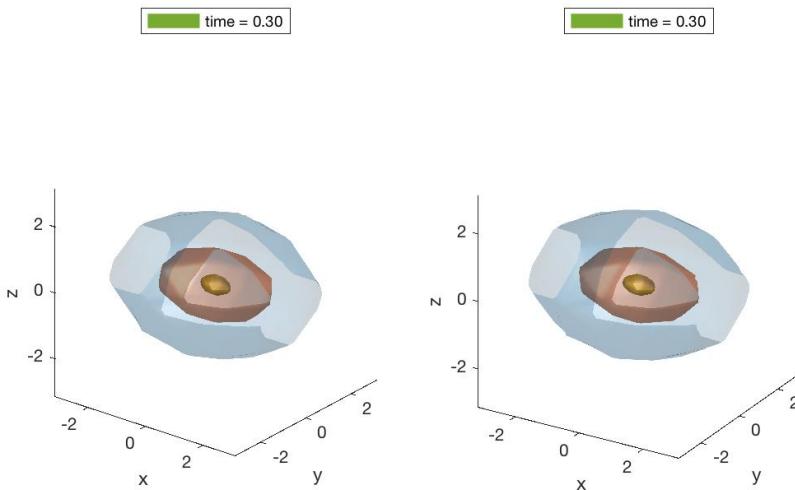
Find which elements of the 8D array are interior micro-grid points and hence correspond to dynamical variables.

```

205 u0 = zeros([nSubP,1,nEnsem,nPatch]);
206 u0([1 end],:,:,:)=nan;
207 u0(:,:,1,:)=nan;
208 u0(:,:,1,:)=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 $u = 0.1, 0.3, \dots, 0.9$, 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, ....

228 maxords=6;
229 for ord=0:2:maxords
    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_{ijk}^x , c_{ijk}^y and c_{ijk}^z , 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
25
26 Microscale space-steps. Q: is using i,j,k slower than 2:end-1??
27
28 dx = diff(patches.x(2:3)); % x micro-scale step
29 dy = diff(patches.y(2:3)); % y micro-scale step

```

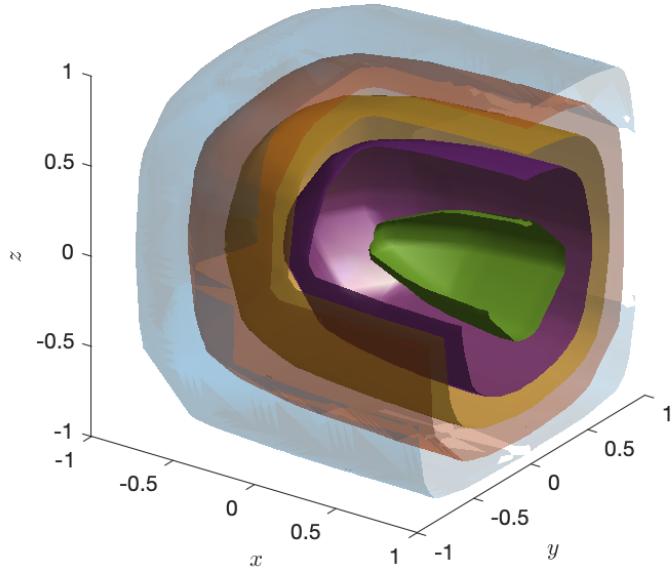
```
33     dz = diff(patches.z(2:3)); % z micro-scale step
34     i = 2:size(u,1)-1; % x interior points in a patch
35     j = 2:size(u,2)-1; % y interior points in a patch
36     k = 2:size(u,3)-1; % z interior points in a patch
```

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)`

```
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.

```

59 nSubP = mPeriod+2;
60 nPatch = 5;
61 Dom.type = 'equispace';
62 Dom.bcOffset = zeros(2,3); Dom.bcOffset(2) = 0.5;
63 configPatches3(@microDiffBdry3, [-1 1], Dom ...
64 , nPatch, 0, 0.05, nSubP, 'EdgyInt',true ...
65 , 'hetCoeffs',cDiff );

```

Set forcing, and store in global patches for access by the microcode

```

73 patches.fu = 10*exp(-patches.x.^2-patches.y.^2-patches.z.^2);
74 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.

```

87 u0 = zeros([nSubP,1,1,nPatch,nPatch,nPatch]);
88 u0([1 end],:,:, :) = nan;
89 u0(:,[1 end],:,:) = nan;
90 u0(:,:, [1 end],:) = nan;
91 patches.i = find(~isnan(u0));
92 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 3.17).

```

101 disp('Solving system, takes 10--40 secs'),tic
102 uSoln = fsolve(@theRes,u0(patches.i) ...
103 ,optimoptions('fsolve','Display','off'));
104 solveTime = toc
105 normResidual = norm(theRes(uSoln))
106 normSoln = norm(uSoln)

```

Store the solution into the patches, and give magnitudes.

```

112 u0(patches.i) = uSoln;
113 u0 = patchEdgeInt3(u0);

```

Plot isosurfaces of the solution

```

122 figure(1), clf
123 rgb=get(gca,'defaultAxesColorOrder');

```

Reshape spatial coordinates of patches.

```

129 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.

```

137 u = reshape( permute(squeeze(u0),[2 5 1 4 3 6]) ...
138     , [numel(y) numel(x) numel(z)]);
139 maxu=max(u(:)), minu=min(u(:))

    Optionally cut-out the front corner so we can see inside.

145 u( (x'>0) & (y<0) & (shiftdim(z,-2)>0) ) = nan;

    Draw some isosurfaces.

151 clf;
152 for iso=5:-1:1
153     isov=(iso-0.5)/5*(maxu-minu)+minu;
154     hsurf(iso) = patch(isosurface(x,y,z,u,isov));
155     isonormals(x,y,z,u,hsurf(iso))
156     set(hsurf(iso) , 'FaceColor',rgb(iso,:)
157         , 'EdgeColor','none' , 'FaceAlpha',iso/5);
158     hold on
159 end
160 hold off
161 axis equal, axis([-1 1 -1 1 -1 1]), view(35,25)
162 xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
163 camlight, lighting gouraud
164 if OurCf2eps(mfilename) %optionally save plot
165 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_{ijk}^x , c_{ijk}^y and c_{ijk}^z , 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.

```

191 function ut = microDiffBdry3(t,u,patches)
192 if nargin<3, global patches, end

```

Microscale space-steps.

```

198 dx = diff(patches.x(2:3)); % x micro-scale step
199 dy = diff(patches.y(2:3)); % y micro-scale step
200 dz = diff(patches.z(2:3)); % z micro-scale step
201 i = 2:size(u,1)-1; % x interior points in a patch
202 j = 2:size(u,2)-1; % y interior points in a patch
203 k = 2:size(u,3)-1; % z 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).

```

211 u( 1 ,:,:,:,:, 1 ,:,:)=0; %left face of leftmost patch
212 u(end,:,:,:,:,:)=u(end-1,:,:,:,:,:); %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 To do

- Wider edges of interpolation—maybe code as reshaping to multiple variables.
- Core averages code.
- Adapt to maps in micro-time? Surely easy, just an example.

5.7 Miscellaneous tests

5.7.1 patchEdgeInt1test: test the 1D patch coupling

A script to test the spectral and finite-order polynomial interpolation of function `patchEdgeInt1()`. Tests one or several variables, normal and staggered grids, and also tests centre and edge interpolation. But does not yet test core averaging, nor divided differences on staggered, etc.

Start by establishing global data struct for the range of various cases.

```
22 clear all
23 global patches
24 nSubP=5
```

5.7.1.1 Check divided difference interpolation

But not yet implemented staggered grid version?? Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The `@sin` is a dummy.)

```
41 for edgyInt=[mod(nSubP,2)==0 true]
42 for ordCC=2:2:8
43 for nPatch=ordCC+(2:4)
44     edgyInt=edgyInt
45     ordCC=ordCC
46     nPatch=nPatch
47     Domain=5*[-rand rand]
48     ratio=0.5*rand
49     configPatches1(@sin,Domain,nan,nPatch,ordCC,ratio,nSubP ...
50         ,’EdgyInt’,edgyInt);
```

Until `configPatches1` updated, change the data structure here: first, specify general divided differences.

```
57 patches.periodic=false;
```

Second, displace patches to a random non-uniform spacing.

```
63 H = diff(patches.x(1,:,:,:1:2));
64 patches.x = patches.x+0.8*H*(rand(1,1,1,nPatch)-0.5);
65 H = squeeze( diff(patches.x(1,:,:,:)) )% for information only
```

Check multiple fields simultaneously Set profiles to be various powers of x , ps , and store as different ‘variables’ at each point.

```
76     ps=1:ordCC
77     cs=randn(size(ps))
78     u0=patches.x.^ps.*cs+randn;
```

Then evaluate the interpolation and squeeze the singleton dimension of an ‘ensemble’.

```

85      ui=patchEdgeInt1(u0(:));
86      ui=squeeze(ui);

```

All patches should have zero error: but need to either in `patchEdgeInt1` comment out NaN assignment of boundary values, or not test the two extreme patches here, or add code to omit NaNs here. High-order interpolation seems to be more affected by round-off so relax error size.

```

96      j=1:nPatch;
97      iError=ui(:,:,j)-u0(:,:,j);
98      hist(log10(abs(iError(abs(iError)>0))),-17:-10)
99      xlabel('log10 iError'), pause(0.3)%%
100     normError=norm(iError(:))
101     assert(normError<5e-10 ...
102           , 'failed divided difference interpolation')

```

End the for-loops over various parameters.

```

109 end,end,end
110 disp('Passed all divided difference interpolation')
111 %error('Not testing any further')% temporary halt

```

5.7.1.2 Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Say do three realisations for each number of patches.

```

130 nReal=3
131 for nPatch=repmat(5:10,1,nReal)
132 nPatch=nPatch
133 Len=10*rand
134 ratio=0.5*rand
135 configPatches1(@sin,[0,Len],nan,nPatch,0,ratio,nSubP ...
136     , 'EdgyInt',rand<0.5); % random Edgy or not
137 if mod(nPatch,2)==0, disp('Avoiding highest wavenumber'), end
138 kMax=floor((nPatch-1)/2);
139 if patches.EdgyInt, i0=[2 nSubP-1], else i0=(nSubP+1)/2, end
140 patches.periodic=true; % temporary

```

Test single field Set a profile, and evaluate the interpolation.

```

148 for k=-kMax:kMax
149     u0=exp(1i*k*patches.xx*2*pi/Len);
150     ui=patchEdgeInt1(u0(:));
151     normError=rms(ui(:)-u0(:));
152     if abs(normError)>5e-14
153         normError=normError, k=k
154         error(['failed single var interpolation k=' num2str(k)])
155     end
156 end

```

Test multiple fields Use this to measure some of the errors in order to omit singleton dimensions, and also squish any errors if the second argument is essential zero (to cater for cosine aliasing errors).

```
167 normDiff=@(u,v) ...
168     norm(squeeze(u)-squeeze(v))*norm(squeeze(v(i0,:,:,:)));
```

Set a profile, and evaluate the interpolation. For the case of the highest wavenumber, squash the error when the centre-patch values are all zero by multiplying by result norm. Not yet working for edgy interpolation.

```
178 for k=1:(nPatch-1)/2 % not checking the highest wavenumber
179     u0=sin(k*patches.x*2*pi/Len);
180     v0=cos(k*patches.x*2*pi/Len);
181     uvi=patchEdgeInt1( reshape([u0 v0],[],1) );
182     normuError=normDiff(uvi(:,1,:,:),u0);
183     normvError=normDiff(uvi(:,2,:,:),v0);
184     if abs(normuError)+abs(normvError)>2e-13
185         normuError=normuError, normvError=normvError
186         error(['failed double field interpolation k=' num2str(k)])
187     end
188 end
189
190 End the for-loop over various geometries.
191
192 end
193 disp('Passed standard spectral interpolation tests')
```

5.7.1.3 Now test spectral interpolation on staggered grid

Must have even number of patches for a staggered grid.

```
212 disp('*** spectral interpolation on staggered grid')
213 for nPatch=repmat(6:2:20,1,nReal)
214     nPatch=nPatch
215     ratio=0.5*rand
216     nSubP=7; % of form 4*N-1
217     Len=10*rand
218     configPatches1(@simpleWavePde,[0,Len],nan,nPatch,-1,ratio,nSubP ...
219         ,,'EdgyInt',rand<0.5);
220     if mod(nPatch,4)==0, disp('Avoiding highest wavenumber'), end
221     kMax=floor((nPatch/2-1)/2)
222     if patches.EdgyInt, i0=[2 nSubP-1], else i0=(nSubP+1)/2, end%??
223     patches.periodic=true; % temporary
```

Identify which microscale grid points are h or u values.

```
229 uPts=mod( (1:nSubP)+(1:nPatch) ,2);
230 hPts=find(1-uPts);
231 uPts=find(uPts);
```

Set a profile for various wavenumbers. The capital letter U denotes an array of values merged from both u and h fields on the staggered grids.

```

239 fprintf('Staggered: single field-pair test.\n')
240 for k=-kMax:kMax
241     U0=nan(nSubP,nPatch);
242     U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
243     U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
244     Ui=patchEdgeInt1(U0(:));
245     normError=norm(Ui(:)-U0(:));
246     if abs(normError)>5e-14
247         normError=normError
248         patches=patches
249         error(['staggered: failed single sys interpolation k=' num2str(k)])
250     end
251 end

```

Test multiple fields Use this to measure some of the errors in order to omit singleton dimensions, and also squish any errors if the third argument is essential zero (to cater for cosine aliasing errors).

```

262 normDiff=@(u,v,w) ...
263 norm(squeeze(u)-squeeze(v))*norm(squeeze(w(i0,:,:,:)));

```

Set a profile, and evaluate the interpolation. For the case of the highest wavenumber zig-zag, squash the error when the alternate centre-patch values are all zero. First shift the x -coordinates so that the zig-zag mode is centred on a patch.

```

273 fprintf('Staggered: Two field-pairs test.\n')
274 x0=patches.x((nSubP+1)/2,1);
275 patches.x=patches.x-x0;
276 oddP=1:2:nPatch; evnP=2:2:nPatch;
277 for k=1:kMax
278     U0=nan(nSubP,1,1,nPatch); V0=U0;
279     U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
280     U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
281     V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
282     V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
283     UVi=patchEdgeInt1([U0 V0]);
284 %   U0i0odd=squeeze(U0(i0,:,:,:,oddP)), U0i0evn=squeeze(U0(i0,:,:,:,evnP))
285 %   V0i0odd=squeeze(V0(i0,:,:,:,oddP)), V0i0evn=squeeze(V0(i0,:,:,:,evnP))
286     normuError=[normDiff(UVi(:,1,:,:,:,oddP),U0(:,:,1,:,oddP),U0(:,:,1,:,evnP))
287                 normDiff(UVi(:,1,:,:,:,evnP),U0(:,:,1,:,evnP),U0(:,:,1,:,oddP))]';
288     normvError=[normDiff(UVi(:,2,:,:,:,oddP),V0(:,:,2,:,oddP),V0(:,:,2,:,evnP))
289                 normDiff(UVi(:,2,:,:,:,evnP),V0(:,:,2,:,evnP),V0(:,:,2,:,oddP))]';
290     if norm(normuError)+norm(normvError)>2e-13
291         normuError=normuError, normvError=normvError
292         [U0 UVi(:,1,:,:,:) V0 UVi(:,2,:,:,:)]
293         patches=patches
294         error(['staggered: failed double field interpolation k=' num2str(k)])
295     end
296 end

```

End for-loop over patches

```
303 end
```

5.7.1.4 Check standard finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```
320 for edgyInt=[mod(nSubP,2)==0 true]
321 for ordCC=2:2:8
322 for nPatch=ordCC+(2:4)
323   edgyInt=edgyInt
324   ordCC=ordCC
325   nPatch=nPatch
326   Domain=5*[-rand rand]
327   ratio=0.5*rand
328   configPatches1(@sin,Domain,nan,nPatch,ordCC,ratio,nSubP ...
329     , 'EdgyInt',edgyInt);
330   patches.periodic=true; % temporary
```

Check multiple fields simultaneously Set profiles to be various powers of x , ps , and store as different ‘variables’ at each point.

```
339 ps=1:ordCC
340 cs=randn(size(ps))
341 u0=patches.x.^ps.*cs+randn;
```

Then evaluate the interpolation and squeeze the singleton dimension of an ‘ensemble’.

```
348 ui=patchEdgeInt1(u0(:));
349 ui=squeeze(ui);
```

The interior patches should have zero error.

```
355 j=ordCC/2+1:nPatch-ordCC/2;
356 iError=ui(:,:,j)-u0(:,:,j);
357 normError=norm(iError(:))
358 assert(normError<5e-12 ...
359   , 'failed finite stencil interpolation')
```

End the for-loops over various parameters.

```
366 end,end,end
367 disp('Passed all standard polynomial interpolation')
```

5.7.2 patchEdgeInt2test: tests 2D patch coupling

A script to test the spectral, finite-order, and divided difference, polynomial interpolation of function `patchEdgeInt2()`. Tests one or several variables, normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc as they are not yet implemented.

Start by establishing global data struct for the range of various cases. Choose a number of realisations for every type.

```
21 clear all, close all
22 global patches
23 nRealise = 19
```

5.7.2.1 Check divided difference interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The @sin is a dummy.)

```
40 maxErrors=[];
41 for realisation = 1:nRealise
42     Lx = 1+3*rand; Ly = 1+3*rand;
43     Domain = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
44     nSubP = 1+2*randi(3,1,2)
45     nPatch = randi([3 8],1,2)
46     dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
47     ordCC = 2*randi([1 4])
48     edgyInt = (rand>0.5)
49     configPatches2(@sin,Domain,'equispace' ...
50                 ,nPatch,ordCC,dx,nSubP,'EdgyInt',edgyInt);
```

Second, displace patches to a random non-uniform spacing.

```
56 Hx = diff(patches.x(1,1,:,:,:,1:2,1));
57 patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,nPatch(1),1)-0.5);
58 Hx = squeeze( diff(patches.x(1,1,:,:,:,1)) );% for information only
59 Hy = diff(patches.y(1,1,:,:,:,1:2));
60 patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,nPatch(2))-0.5);
61 Hy = squeeze( diff(patches.y(1,1,:,:,:,1,:)) );% for information only
```

Check multiple fields simultaneously Set profiles to be various powers of x and y , ps and qs , and store as different ‘variables’ at each point. First, limit the order of test polynomials by the order of interpolation and by the number of patches.

```
74 ox=min(ordCC,nPatch(1)-1);
75 oy=min(ordCC,nPatch(2)-1);
76 [ps,qs]=ndgrid(0:ox,0:oy);
77 ps=reshape(ps,1,1,[ ]);
78 qs=reshape(qs,1,1,[ ]);
79 cs=2*rand(size(ps))-1;
80 u0=cs.*patches.x.^ps.*patches.y.^qs;
81 % sizeu0=size(u0)
```

Then evaluate the interpolation

```
87 u=u0; u([1 end],:)=inf; u(:,[1 end],:)=inf;
88 ui=patchEdgeInt2(u(:));
```

All patches should have zero error: but need to either in `patchEdgeInt2` comment out NaN assignment of boundary values, or not test the two extreme patches here, or add code to omit NaNs here. High-order interpolation seems to be more affected by round-off so relax error size.

```

98     error = ui-u0;
99     % hist(log10(abs(error(abs(error)>1e-20))),-20:-8)
100    % xlabel('log10 error')
101    % pause(0.1)%%
102    maxError=max(abs(error(:)))
103    maxErrors=[maxErrors maxError];
104    assert(maxError<1e-9 ...
105        , 'failed divided difference interpolation')
106    disp('***** This divided difference test passed')

```

End the for-loops over various parameters.

```

113 end% for realisation
114 maxMaxErrorDividedDiff = max(maxErrors)
115 disp('***** Passed all divided difference interpolation')
116 pause(1)

```

5.7.2.2 Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
132 for realisation=1:nRealise
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```

140 Lx = 1+3*rand, Ly = 1+3*rand
141 nSubP = 1+2*randi(3,1,2)
142 ratios = rand(1,2)/2
143 nPatch = randi([3 6],1,2)
144 edgyInt = (rand>0.5)
145 configPatches2(@sin,[0 Lx 0 Ly],nan ...
146     ,nPatch,0,ratios,nSubP,'EdgyInt',edgyInt);

```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```

156 nV=randi(3)
157 [nx,Nx]=size(squeeze(patches.x));
158 [ny,Ny]=size(squeeze(patches.y));
159 u0=nan(nx,ny,nV,1,Nx,Ny);
160 for iV=1:nV
161     kx=randi([0 floor((nPatch(1)-1)/2)])
162     ky=randi([0 floor((nPatch(2)-1)/2)])

```

```

163     phix=pi*rand*(2*kx~=nPatch(1))
164     phiy=pi*rand*(2*ky~=nPatch(2))
165     % generate 6D array via auto-replication
166     u0(:,:,iV,1,:,:)=sin(2*pi*kx*patches.x/Lx+phix) ...
167             .*sin(2*pi*ky*patches.y/Ly+phiy);
168 end

```

Copy and NaN the edges, then interpolate

```

174 u=u0; u([1 end],:,:)=nan; u(:,[1 end],:)=nan;
175 u=patchEdgeInt2(u(:));

```

Compute difference, ignoring the nans which should only be in the corners. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```

184 error=u-u0;
185 assert(all(~isnan(error(:))),'found nans in the error!')
186 normError=norm(error(:))
187 assert(normError<1e-12, '2D spectral interpolation failed')
188 disp('***** This spectral test passed')

```

End the for-loop over realisations

```

195 end
196 disp('***** All the spectral tests passed')
197 pause(1)

```

5.7.2.3 Check standard finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The @sin is a dummy.)

```

215 for realisations=1:nRealise
216     ordCC=2*randi([1 4])
217     nPatch=ordCC+randi([2 4],1,2)
218     edgyInt = (rand>0.5)
219     nSubP = 1+2*randi(3,1,2)
220     Domain=5*[-rand rand -rand rand]
221     ratios=0.5*rand(1,2)
222     configPatches2(@sin,Domain,nan ...
223                 ,nPatch,ordCC,ratios,nSubP,'EdgyInt',edgyInt);

```

Check multiple fields simultaneously Set profiles to be various powers of x , ps , and store as different ‘variables’ at each point.

```

232 [ps,qs]=meshgrid(0:ordCC);
233 ps=reshape(ps,1,1,[]); qs=reshape(qs,1,1,[]);
234 u0=ones(size(ps)).*patches.x.^ps.*patches.y.^qs;

```

Then evaluate the interpolation.

```

240 ui=patchEdgeInt2(u0(:));

```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```

248     I=ordCC/2+1:nPatch(1)-ordCC/2;
249     J=ordCC/2+1:nPatch(2)-ordCC/2;
250     error=ui(:,:, :, I, J)-u0(:,:, :, I, J);
251     assert(all(~isnan(error(:))), 'found nans in the error!')
252     normError=norm(error(:))
253     assert(normError<5e-8 ...
254     , 'failed finite stencil interpolation')
255     disp('***** This finite stencil test passed')

    End the for-loops over various parameters.

262 end %for realisations
263 disp('***** Passed all standard polynomial interpolation')

```

5.7.2.4 Finished

If no error messages, then all OK.

```
277 disp('***** All the interpolation tests successful')
```

5.7.3 patchEdgeInt3test: tests 3D patch coupling

A script to test the spectral, finite-order, and divided difference, polynomial interpolation of function `patchEdgeInt3()`. Tests one or several variables, normal grids, and also tests centre and edge interpolation. But does not yet test staggered grids, core averaging, etc as they are not yet implemented.

Start by establishing global data struct for the range of various cases. Choose a number of realisations for every type.

```

20 clear all, close all
21 global patches
22 nRealise = 19

```

5.7.3.1 Check divided difference interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths, random ratios, and randomised distribution of patches. (The `@sin` is a dummy.)

```

40 maxErrors=[];
41 for realisation = 1:nRealise
42     Lx = 1+3*rand; Ly = 1+3*rand; Lz = 1+3*rand;
43     Domain = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
44     nSubP = 1+2*randi(3,1,3)
45     nPatch = randi([3 8],1,3)
46     dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
47     ordCC = 2*randi([1 4])
48     edgyInt = (rand>0.5)

```

```

49 configPatches3(@sin,Domain,'equispace' ...
50 ,nPatch,ordCC,dx,nSubP,'EdgyInt',edgyInt);

Second, displace patches to a random non-uniform spacing.

56 Hx = diff(patches.x(1,1,1,:,:,:1:2,1,1));
57 patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,1,nPatch(1),1,1)-0.5);
58 Hx = squeeze( diff(patches.x(1,1,1,:,:,:1,1)) );% for information only
59 Hy = diff(patches.y(1,1,1,:,:,:1,1:2,1));
60 patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,1,nPatch(2),1)-0.5);
61 Hy = squeeze( diff(patches.y(1,1,1,:,:,:1,:1)) );% for information only
62 Hz = diff(patches.z(1,1,1,:,:,:1,1,1:2));
63 patches.z = patches.z+0.8*Hz*(rand(1,1,1,1,1,1,1,nPatch(3))-0.5);
64 Hz = squeeze( diff(patches.z(1,1,1,:,:,:1,1,:)) );% for information only

```

Check multiple fields simultaneously Set profiles to be various powers of x , y and z , **ps**, **qs** and **rs**, and store as different ‘variables’ at each point. First, limit the order of test polynomials by the order of interpolation and by the number of patches.

```

78 ox=min(ordCC,nPatch(1)-1);
79 oy=min(ordCC,nPatch(2)-1);
80 oz=min(ordCC,nPatch(3)-1);
81 [ps,qs,rs]=ndgrid(0:ox,0:oy,0:oz);
82 ps=reshape(ps,1,1,1,[]);
83 qs=reshape(qs,1,1,1,[]);
84 rs=reshape(rs,1,1,1,[]);
85 cs=2*rand(size(ps))-1;
86 u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;

```

Then evaluate the interpolation, setting faces to **inf** for error checking.

```

93 u=u0; u([1 end],:)=inf; u(:,[1 end],:)=inf; u(:,:,1 end,:)=inf;
94 ui=patchEdgeInt3(u(:));

```

All patches should have zero error: but need to either in **patchEdgeInt3** comment out **NaN** assignment of boundary values, or not test the two extreme patches here, or add code to omit **NANs** here. High-order interpolation seems to be more affected by round-off so relax error size.

```

104 error = ui-u0;
105 % optional histogram of errors
106 % hist(log10(abs(error(abs(error)>1e-20))),-20:-8)
107 % xlabel('log10 error')
108 % pause(0.1)%%
109 maxError=max(abs(error(:)))
110 maxErrors=[maxErrors maxError];
111 assert(maxError<1e-7 ...
112 , 'failed divided difference interpolation')
113 disp('***** This divided difference test passed')

```

End the for-loops over various parameters.

```

120 end% for realisation
121 disp('***** Passed all divided difference interpolation')
122 maxMaxErrorDividedDiffs = max(maxErrors)
123 pause(1)

```

5.7.3.2 Test standard spectral interpolation

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
136 for realisation=1:nRealise
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches, randomly edge or mid-patch interpolation.

```

144 Lx = 1+3*rand, Ly = 1+3*rand, Lz = 1+3*rand
145 nSubP = 1+2*randi(3,1,3)
146 ratios = 0.5*rand(1,3)
147 nPatch = randi([3 6],1,3)
148 edgyInt = (rand>0.5)
149 configPatches3(@sin,[0 Lx 0 Ly 0 Lz],nan ...
150 ,nPatch,0,ratios,nSubP,'EdgyInt',edgyInt);

```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift. But if an even number of patches in either direction, then do not test the highest wavenumber because of aliasing problem.

```

160 nV=randi(3)
161 [nx,Nx]=size(squeeze(patches.x));
162 [ny,Ny]=size(squeeze(patches.y));
163 [nz,Nz]=size(squeeze(patches.z));
164 u0=nan(nx,ny,nz,nV,1,Nx,Ny,Nz);
165 for iV=1:nV
166     ks=floor( rand(1,3).*floor(([Nx Ny Nz]+1)/2) )
167     phis = pi*rand(1,3).*(2*ks-[Nx Ny Nz])
168     % generate 8D array via auto-replication
169     u0(:,:,:,:,iV,1,:,:,:)=sin(2*pi*ks(1)*patches.x/Lx+phis(1)) ...
170                 .*sin(2*pi*ks(2)*patches.y/Ly+phis(2)) ...
171                 .*sin(2*pi*ks(3)*patches.z/Lz+phis(3));
172 end

```

Copy and NaN the faces, then interpolate

```

178 u=u0;
179 u([1 end],:,:,:)=nan; u(:,[1 end],:,:)=nan; u(:,:,1,:)=nan;
180 u=patchEdgeInt3(u(:));

```

Compute difference, ignoring the nans which should only be in the corners. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```

189 error=u-u0;
190 assert(all(~isnan(error(:))), 'found nans in the error!')
191 normError=norm(error(:))
192 assert(normError<1e-10, '3D spectral interpolation failed')
193 disp('***** This spectral test passed')

    End the for-loop over realisations

200 end
201 disp('***** All the spectral tests passed')
202 pause(1)

```

5.7.3.3 Check polynomial finite width interpolation

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The `@sin` is a dummy.)

```

220 for realisations=1:nRealise
221     ordCC=2*randi([1 3])
222     nPatch=ordCC+randi([1 4],1,3)
223     edgyInt = (rand>0.5)
224     nSubP = 1+2*randi(3,1,3)
225     Domain=5*[-rand rand -rand rand -rand rand]
226     ratios=0.5*rand(1,3)
227     configPatches3(@sin,Domain,nan ...
228                 ,nPatch,ordCC,ratios,nSubP,'EdgyInt',edgyInt);

```

Check multiple fields simultaneously Set profiles to be various powers of x, y, z , namely `ps`, `qs`, `rs`, and store as different ‘variables’ at each point.

```

238 [ps,qs,rs]=meshgrid(0:ordCC);
239 ps=reshape(ps,1,1,1,[]);
240 qs=reshape(qs,1,1,1,[]);
241 rs=reshape(rs,1,1,1,[]);
242 u0=ones(size(ps)).*patches.x.^ps ...
243             .*patches.y.^qs.*patches.z.^rs;

```

Then evaluate the interpolation.

```
249 ui=patchEdgeInt3(u0(:));
```

The interior patches should have zero error. Appear to need error tolerance of 10^{-8} because of the size of the domain and the high order of interpolation.

```

257 I=ordCC/2+1:nPatch(1)-ordCC/2;
258 J=ordCC/2+1:nPatch(2)-ordCC/2;
259 K=ordCC/2+1:nPatch(3)-ordCC/2;
260 error=ui(:,:, :, :, I, J, K)-u0(:,:, :, :, I, J, K);
261 assert(all(~isnan(error(:))), 'found nans in the error!')
262 normError=norm(error(:))
263 assert(normError<5e-8 ...
264         , 'failed finite stencil polynomial interpolation')
265 disp('***** This finite stencil test passed')

```

End the for-loops over various parameters.

```
272 end%for realisation  
273 disp('***** Passed all polynomial interpolation tests')
```

5.7.3.4 Finished

If no error messages, then all OK.

```
287 disp('***** All types of interpolation tests successful')
```

6 Matlab parallel computation of the patch scheme

Chapter contents

6.1	<code>chanDispSpmd</code> : simulation of a 1D shear dispersion via simulation on small patches across a channel	257
6.1.1	Simulate heterogeneous advection-diffusion	259
6.1.2	Plot the solution	261
6.1.3	<code>microBurst</code> function for Projective Integration	262
6.1.4	<code>chanDispMicro()</code> : heterogeneous 2D advection-diffusion in a long thin channel	263
6.2	<code>rotFilmSpmd</code> : simulation of a 2D shallow water flow on a rotating heterogeneous substrate	265
6.2.1	Simulate heterogeneous advection-diffusion	266
6.2.2	Plot the solution	269
6.2.3	<code>microBurst</code> function for Projective Integration	270
6.2.4	<code>rotFilmMicro()</code> : 2D shallow water flow on a rotating heterogeneous substrate	271
6.3	<code>homoDiff31spmd</code> : computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion	273
6.3.1	Simulate heterogeneous diffusion	274
6.3.2	Plot the solution	276
6.3.3	<code>microBurst</code> function for Projective Integration	277
6.4	<code>RK2mesoPatch()</code>	279
6.5	To do	283

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 than 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 too 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

6.1.1	Simulate heterogeneous advection-diffusion	259
6.1.2	Plot the solution	261
6.1.3	microBurst function for Projective Integration	262
6.1.4	chanDispMicro(): heterogeneous 2D advection-diffusion in a long thin channel	263

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

$$\frac{\partial c}{\partial t} = -\text{Pe} u(y) \frac{\partial c}{\partial x} + \frac{\partial}{\partial x} \left[\kappa_x(y) \frac{\partial c}{\partial x} \right] + \frac{\partial}{\partial y} \left[\kappa_y(y) \frac{\partial c}{\partial y} \right]. \quad (6.1)$$

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

$$\begin{aligned} \frac{\partial c_{ij}}{\partial t} &= -\text{Pe} u(y_j) \frac{c_{i+1,j} - c_{i-1,j}}{2\delta x} + \frac{d_{i,j+1/2} - d_{i,j-1/2}}{\delta y} + \frac{D_{i+1/2,j} - D_{i-1/2,j}}{\delta x}, \\ d_{ij} &:= \kappa_y(y_j) \frac{c_{i,j+1/2} - c_{i,j-1/2}}{\delta y}, \quad D_{ij} := \kappa_x(y_j) \frac{c_{i+1/2,j} - c_{i-1/2,j}}{\delta x}. \end{aligned} \quad (6.2)$$

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.

```
75  clear all
76  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.

```
86  ny = 7
87  y = linspace(-1,1,2*ny+1);
88  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.

```
101 mPeriod = 4
102 cHetr = shiftdim([3/2 1],-1).*(1-y.^2) ...
103 .*exp(0.3*randn([mPeriod 2*ny+1 2]));
```

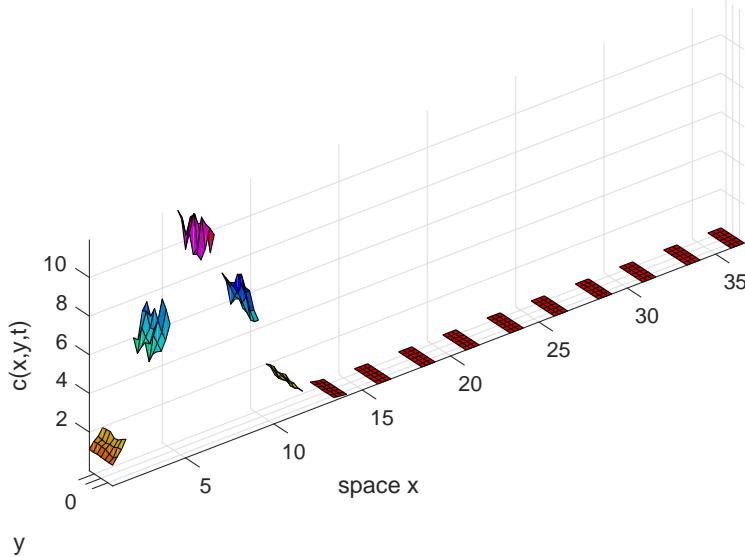
Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Choose some random order of interpolation to see the alternatives. Set `patches` information to be global so the info can be used for Cases 1–2 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
118 if theCase<=2, global patches, end
119 nPatch=15
120 nSubP=2+mPeriod
121 ratio=0.2+0.2*(theCase<4)
122 Len=nPatch/ratio
123 ordCC=2*randi([0 3])
124 disp('**** Setting configPatches1')
125 patches = configPatches1(@chanDispMicro, [0 Len], nan ...
126 , nPatch, ordCC, ratio, nSubP, 'EdgyInt',true ...
127 , '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).

```
135 Peclet = 10
136 if theCase==1, patches.Pe = Peclet;
137 else      spmd, patches.Pe = Peclet; end
138 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
```

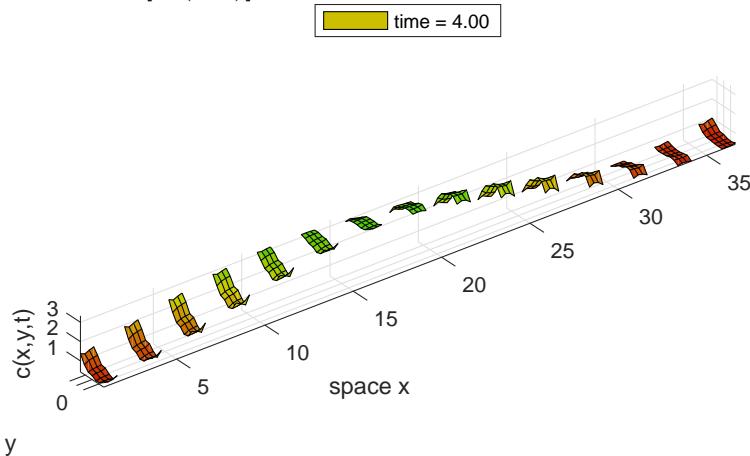
At each time, squeeze sub-patch data into a 3D array, permute to get all the x -variation in the first two dimensions, and reshape into x -variation for each and every (y).

```
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 .

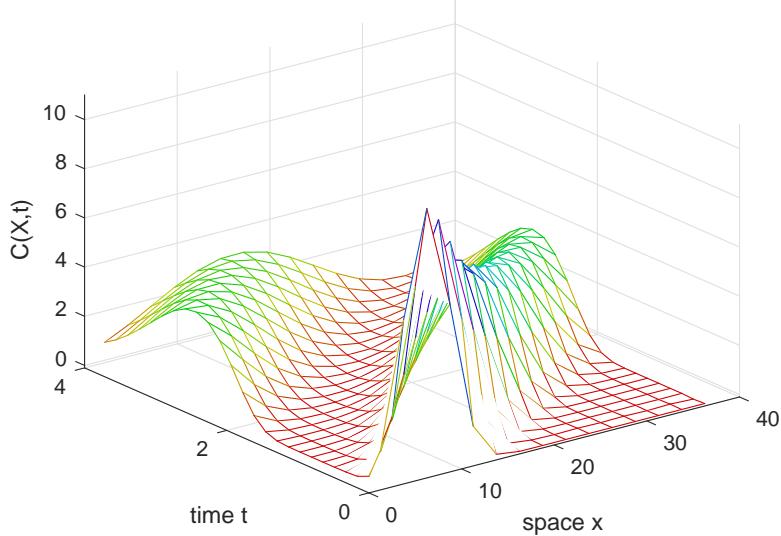
```
374 figure(2), clf, colormap(0.8*hsv)
375 X = squeeze(mean(x(2:end-1,:,:,:)));
376 C = squeeze(mean(mean(cs(:,2:end-1,:,:,:),2),3));
377 j = 1:ceil(nTimes/30):nTimes;
378 mesh(X,ts(j),C(j,:));
379 xlabel('space x'), ylabel('time t'), zlabel('C(X,t)')
380 zlim([-0.1 11])
381 ifOurCf2eps([mfilename '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 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 advects 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

6.2.1	Simulate heterogeneous advection-diffusion	266
6.2.2	Plot the solution	269
6.2.3	microBurst function for Projective Integration	270
6.2.4	rotFilmMicro(): 2D shallow water flow on a rotating heterogeneous substrate	271

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.)

$$\frac{\partial h}{\partial t} = -\nabla \cdot (h\vec{v}), \quad (6.3a)$$

$$\frac{\partial \vec{v}}{\partial t} = \begin{bmatrix} -b & f \\ -f & -b \end{bmatrix} \vec{v} - (\vec{v} \cdot \nabla) \vec{v} - g\nabla h + \vec{\nabla} \cdot (\nu \vec{\nabla} \vec{v}), \quad (6.3b)$$

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;
```

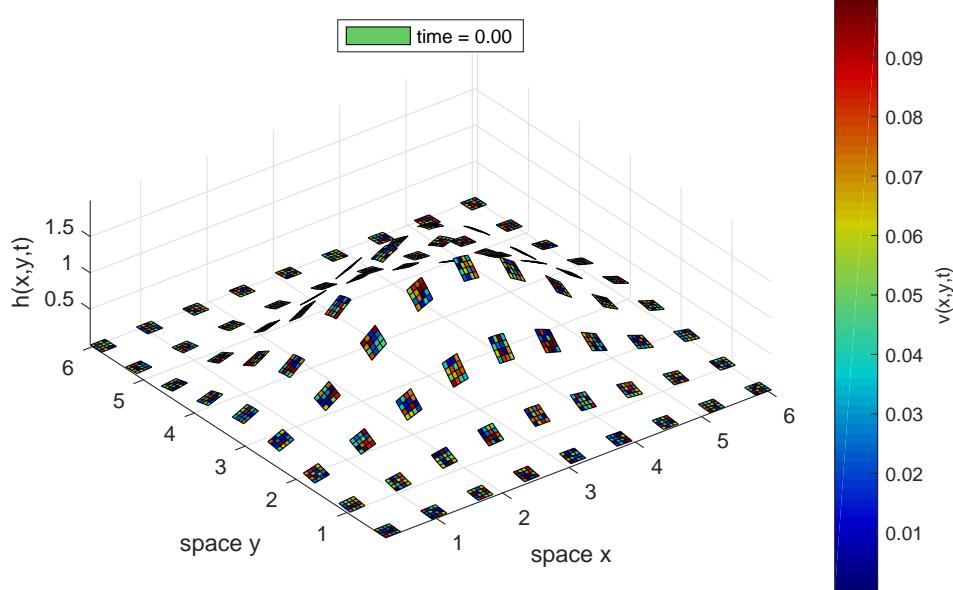
Go to the selected case.

```
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,[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 then reshape to 2D.

```

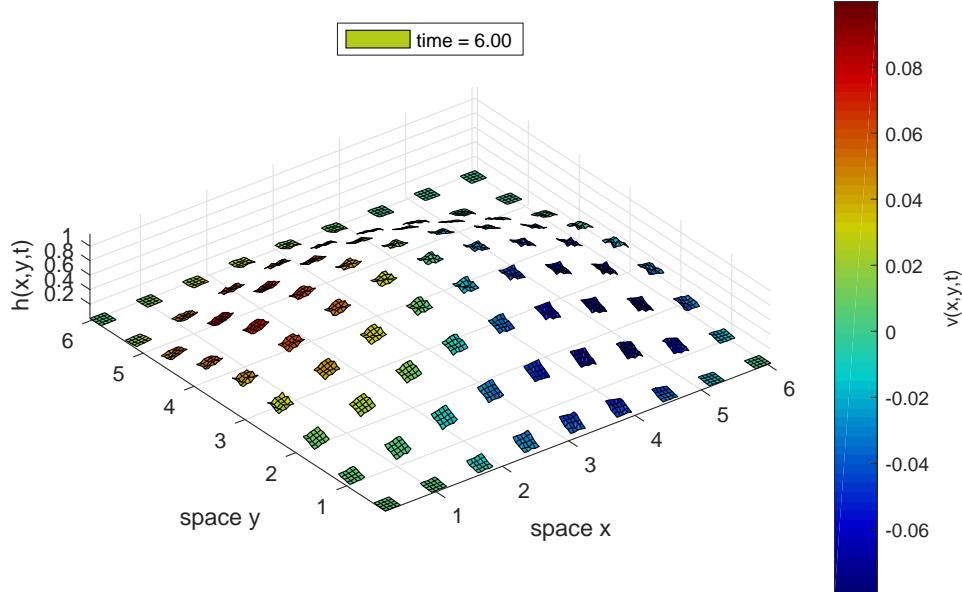
325 h = reshape( permute( squeeze( ...
326     huvs(l,:,:,:,1,1,:,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
327 u = reshape( permute( squeeze( ...
328     huvs(l,:,:,:,2,1,:,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
329 v = reshape( permute( squeeze( ...
330     huvs(l,:,:,:,3,1,:,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));

```

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
370 end%for over time
371 ifOurCf2eps([mfilename 'tFin'])

```

Finish the animation loop, and optionally save the final plot to file, [Figure 6.5](#).

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 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 × 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;

```

```

57 u4v = (huv(i,j+1,u,:,:,:)+huv(i-1,j,u,:,:,:))/4;
58     +huv(i,j,u,:,:,:)+huv(i-1,j+1,u,:,:,:))/4;
59 h2u = (huv(2:nx,:,h,:,:,:)+huv(1:nx-1,:,h,:,:,:))/2;
60 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):

```

67 huvt(i,j,h,:,:,:) = ...
68 - (h2u(i,j ,:,:,:,:).*huv(i ,j,u,:,:,:)) ...
69 - h2u(i-1,j ,:,:,:,:).*huv(i-1,j,u,:,:,:)/dx ...
70 - (h2v(i,j ,:,:,:,:).*huv(i,j ,v,:,:,:)) ...
71 - 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):

```

79 huvt(i,j,u,:,:,:) = ...
80 - p.cs(i,j,b).*huv(i,j,u,:,:,:)+ p.f.*v4u ...
81 - huv(i,j,u,:,:,:).*(huv(i+1,j,u,:,:,:)-huv(i-1,j,u,:,:,:))/(2*dx) ...
82 - v4u.* (huv(i,j+1,u,:,:,:)-huv(i,j-1,u,:,:,:))/(2*dy) ...
83 - p.g*(huv(i+1,j,h,:,:,:)-huv(i,j,h,:,:,:))/dx ...
84 + diff(p.cs(:,j,nux).*diff(huv(:,j,u,:,:,:),[],1),[],1)/dx^2 ...
85 + 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):

```

93 huvt(i,j,v,:,:,:) = ...
94 - p.cs(i,j,b).*huv(i,j,v,:,:,:)- p.f.*u4v ...
95 - u4v.* (huv(i+1,j,v,:,:,:)-huv(i-1,j,v,:,:,:))/(2*dx) ...
96 - huv(i,j,v,:,:,:).*(huv(i,j+1,v,:,:,:)-huv(i,j-1,v,:,:,:))/(2*dy) ...
97 - p.g*(huv(i,j+1,h,:,:,:)-huv(i,j,h,:,:,:))/dy ...
98 + diff(p.cs(:,j,nux).*diff(huv(:,j,v,:,:,:),[],1),[],1)/dx^2 ...
99 + diff(p.cs(i,:,nuy).*diff(huv(i,:,v,:,:,:),[],2),[],2)/dy^2 ;
100 end% function

```

6.3 homoDiff31spmd: computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion

Section contents

6.3.1	Simulate heterogeneous diffusion	274
6.3.2	Plot the solution	276
6.3.3	microBurst function for Projective Integration	277

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.

```

73 if any(theCase==[1 2]), global patches, end
74 nSubP=mPeriod+2
75 nPatch=[9 1 1]
76 ratio=0.3
77 Len=nPatch(1)/ratio
78 ordCC=2*randi([0 3])
79 disp('**** Setting configPatches3')
80 patches = configPatches3(@heteroDiff3,[0 Len 0 1 0 1], nan ...
81 , nPatch, ordCC, [ratio 1 1], nSubP, 'EdgyInt',true ...
82 , 'hetCoeffs',cHetr , 'parallel',(theCase>1) );

```

6.3.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in [Figure 6.6](#).

```

92 disp('**** Set initial condition and testing du0dt =')
93 if theCase==1

```

Without parallel processing, invoke the usual operations.

```

99 u0 = exp( -(patches.x-Len/2).^2/Len ...
100 -patches.y.^2/2-patches.z.^2 );
101 u0 = u0.*((1+0.2*rand(size(u0))));
102 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`.

```

115 else, spmd
116     u0 = exp( -(patches.x-Len/2).^2/Len ...
117 -patches.y.^2/2-patches.z.^2/4 );
118     u0 = u0.*((1+0.2*rand(size(u0),patches.codist)));
119     du0dt = patchSys3(0,u0,patches);
120     end%spmd
121 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.

```

144 warning('Integrating system in time, wait patiently')
145 ts=0.4*linspace(0,1,21).^2;

```

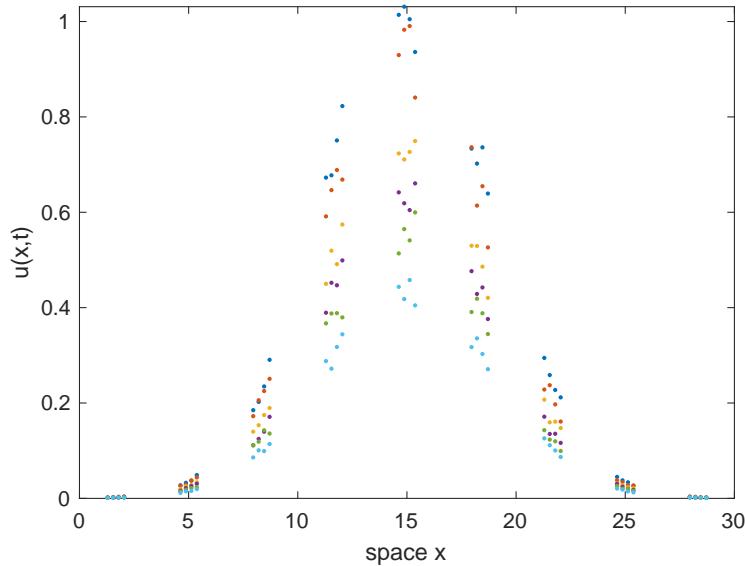
Go to the selected case.

```

151 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.

```

210 case 4
211     microBurst = @(tb0,xb0,bT) ...
212         aBurst(tb0 ,reshape(xb0,size(u0)) ,patches);
213     ts = 0:3:51
214     us = PIRK2(microBurst,ts,gather(u0(:)));
215     us = reshape(us,[length(ts) size(u0)]);
222 end%switch theCase

```

End the four cases.

6.3.2 Plot the solution

Optionally save some plots to file.

```
233 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 .

```

241 figure(1), clf
242 if theCase==1
243     x = reshape( patches.x(2:end-1,:,:,:) ,[],1);
244 else, spmd
245     x = reshape(gather( patches.x(2:end-1,:,:,:) ),[],1);
246 end%spmd
247 x = x{1};
248 end

```

For every time step draw the field values as dots and pause for a short display.

```
255 nTimes = length(ts)
256 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) .

```
265 u = reshape( permute( squeeze( ...
266     us(1,2:end-1,2:end-1,2:end-1,:) ) ,[1 4 2 3]) ,numel(x),[]);
```

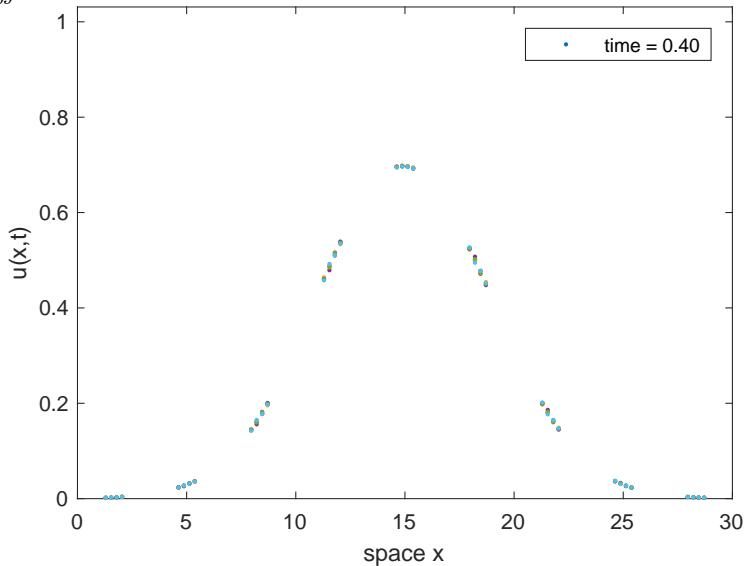
Draw point data to show spread at each cross-section, as well as macro-scale variation in the long space direction.

```
273 if l==1
274     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}^ℓ 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 `configPatchesn` and provided as either as parameter, or as a global variable.

Output

- `xs`, 5/7/9D (depending upon `nD`) array of length $\ell \times \dots$ 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}^ℓ of local error estimate for the step from t_{k-1} to t_k .

Code of RK2 integration Set default number of micro-scale time-steps in each requested meso-scale step of `ts`. Cannot use `nargin` inside explicit `spmd`, but can use it if the `spmd` is already active from the code that invokes this function.

```
79 if nargin<3|isempty(nMicro), nMicro=10; end
```

If patches are set to be in parallel (there must be a parallel pool), but only one worker available (i.e., not already inside `spmd`), then invoke function recursively inside `spmd`. Q: is `numlabs` defined without the parallel computing toolbox??

```

89 if isequal(class(patches), 'Composite') && numlabs==1
90     spmd,
91         [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches);
92     end% spmd
93     assert(isequal(class(xs) , 'Composite'), ' xs  not composite')
94     assert(isequal(class(errs),'Composite'), 'errs not composite')
95     return
96 end

```

Set the number of space dimensions from the number stored patch-size ratios.

```
104 nD = length(patches.ratio);
```

Set the micro-time-steps and create storage for outputs.

```

110 dt = diff(ts)/nMicro;
111 xs = nan([numel(ts) size(x0)]);
112 errs = nan(numel(ts),1);

```

Initialise first result to the given initial condition, and evaluate the initial time derivative into `f1`. Use inter-patch interpolation to ensure edge values of the initial condition are defined and are reasonable.³

```

127 switch nD
128     case 1, x0 = patchEdgeInt1(x0,patches);
129         xs(1,:,:,:,:) = gather(x0);
130     case 2, x0 = patchEdgeInt2(x0,patches);
131         xs(1,:,:,:,:, :) = gather(x0);
132     case 3, x0 = patchEdgeInt3(x0,patches);
133         xs(1,:,:,:,:, :, :) = gather(x0);
134     end;%switch nD
135 errs(1) = 0;
136 f1 = patches.fun(ts(1),x0,patches);

```

Compute the meso-time-steps from t_k to t_{k+1} , copying the derivative `f1` at the end of the last micro-time-step to be the derivative at the start of this one.

```
145 for k = 1:numel(dt)
```

Perform meso-time burst with the new interpolation for edge values, and an interpolation of the time derivatives to get derivative estimates of the edge-values.

³ These `gather()` functions cause all-to-all interprocessor communication once every meso-step. Maybe better to use distributed array instead, (although need to then need to put time index last instead of first??), but we need to do some inter-cpu communication in order to estimate errors.

```

153     switch nD
154         case 1, dx0 = patchEdgeInt1(f1,patches);
155         case 2, dx0 = patchEdgeInt2(f1,patches);
156         case 3, dx0 = patchEdgeInt3(f1,patches);
157     end;%switch nD

```

Perform the micro-time steps.

```

163     for m=1:nMicro
164         f0 = f1;
165         % assert(iscodistributed(f0),'f0 not codist')

```

For all micro-time derivative evaluations, include that the edge values are varying according to the estimate made at the start of the meso-time-step.

```

173     switch nD
174         case 1, f0([1 end],:,:,:,:)=dx0([1 end],:,:,:,:);
175         case 2, f0([1 end],:,:, :, :, :)=dx0([1 end],:,:, :, :, :);
176             f0(:,[1 end],:,:, :, :, :)=dx0(:,[1 end],:,:, :, :, :);
177         case 3
178             f0([1 end],:,:, :, :, :, :, :)=dx0([1 end],:,:, :, :, :, :, :);
179             f0(:,[1 end],:,:, :, :, :, :, :)=dx0(:,[1 end],:,:, :, :, :, :, :);
180             f0(:,:, [1 end],:,:, :, :, :, :, :)=dx0(:,:, [1 end],:,:, :, :, :, :, :);
181     end;%switch nD
182     % assert(iscodistributed(f0),'f0 not codist two')

```

Simple second-order accurate Runge–Kutta micro-scale time-step.

```

189     xh = x0+f0*dt(k)/2;
190     % assert(iscodistributed(xh),'xh not codist')
191     fh = patches.fun(ts(k)+dt(k)*(m-0.5),xh,patches);
192     % assert(iscodistributed(fh),'fh not codist one')
193     switch nD
194         case 1, fh([1 end],:,:,:,:)=dx0([1 end],:,:,:,:);
195         case 2, fh([1 end],:,:, :, :, :)=dx0([1 end],:,:, :, :, :);
196             fh(:,[1 end],:,:, :, :, :)=dx0(:,[1 end],:,:, :, :, :);
197         case 3
198             fh([1 end],:,:, :, :, :, :, :)=dx0([1 end],:,:, :, :, :, :, :);
199             fh(:,[1 end],:,:, :, :, :, :, :)=dx0(:,[1 end],:,:, :, :, :, :, :);
200             fh(:,:, [1 end],:,:, :, :, :, :, :)=dx0(:,:, [1 end],:,:, :, :, :, :, :);
201     end;%switch nD
202     % assert(iscodistributed(fh),'fh not codist two')
203     x0 = x0+fh*dt(k);
204     % assert(iscodistributed(x0),'x0 not codist two')

```

End the burst of micro-time-steps.

```

210     end

```

At the end of each meso-step burst, refresh the interpolate of the edge values, evaluate time-derivative, and temporarily fill-in edges of derivatives (to ensure error estimate is reasonable).

```

219     switch nD
220         case 1, x0 = patchEdgeInt1(x0,patches);
221             xs(k+1,:,:,:,:) = gather(x0);
222         case 2, x0 = patchEdgeInt2(x0,patches);
223             xs(k+1,:,:,:,:,::) = gather(x0);
224         case 3, x0 = patchEdgeInt3(x0,patches);
225             xs(k+1,:,:,:,:,::,:,:) = gather(x0);
226     end;%switch nD
227 % assert(iscodistributed(x0),'x0 not codist three')
228 f1 = patches.fun(ts(k+1),x0,patches);
229 switch nD
230     case 1, f1([1 end],:,:,:,:)=dx0([1 end],:,:,:,:);
231     case 2, f1([1 end],:,:,:,:,::)=dx0([1 end],:,:,:,:,::);
232         f1(:,[1 end],:,:,:,:)=dx0(:,[1 end],:,:,:,:,:);
233     case 3
234         f1([1 end],:,:,:,:,:,:)=dx0([1 end],:,:,:,:,:,:,:);
235         f1(:,[1 end],:,:,:,:,:,:)=dx0(:,[1 end],:,:,:,:,:,:);
236         f1(:,:,1,[1 end],:,:,:,:)=dx0(:,:,1,[1 end],:,:,:,:);
237 end;%switch nD

```

Use the time derivative at t_{k+1} to estimate an error by storing the difference with what Simpson's rule would estimate over the last micro-time step performed.

```

245     f0=f0-2*fh+f1;
246 % assert(iscodistributed(f0),'f2ndDeriv not codist')
247     errs(k+1) = sqrt(gather(mean(f0(:).^2,'omitnan')))*dt(k)/6;
248 end%for-loop
249 end%function

```

End of the function with results returned in `xs` and `errs`.

6.5 To do

- Detailed profiling of the spmd communication to seek better parallelisation.

Appendix A Create, document and test algorithms

- Upon ‘finalising’ a version of the toolbox:
 1. pdflatex and bibtex `Doc/eqnFreeDevMan.tex` to ensure all is documented properly;
 2. execute `bibexport eqnFreeDevMan` to update the local bibliographic data-file;
 3. pdflatex `Doc/eqnFreeUserMan.tex`, several times, to get a shorter and more user friendly version;
 4. replace the root file `eqnFreeUserMan-newest.pdf` by a renamed copy of the new `Doc/eqnFreeUserMan.pdf`
- To create and document the various functions, we adapt an idea due to Neil D. Lawrence of the University of Sheffield in order to interleave MATLAB/Octave code, and its documentation in LaTeX ([Table A.2](#)).
- Each class of toolbox functions is located in separate folders in the repository, say `Dir`.
- Create a LaTeX file `Dir/funs.tex`: establish as one LaTeX chapter that `\input{../Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, [Table A.1](#).
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/docBody.tex` so that people can most easily work on one chapter at a time:
 - create a ‘link’ file `Doc/funs.tex` whose only active content is the command `\input{../Dir/funs.tex}`;
 - put `\include{funs}` into `Doc/docBody.tex`;
 - in `Doc/docBody.tex` modify the `\graphicspath` command to include `{../Dir/Figs}`.
- Each toolbox function is documented as a separate section, within its chapter, with tests and examples as separate sections.
- Each function-section and test-section is to be created as a MATLAB/Octave `Dir/*.m` file, say `Dir/fun1.m`, so that users simply invoke the function in MATLAB/Octave as usual by `fun1(...)`.

Some editors may need to be told that `fun1.m` is a LaTeX file. For example, TexShop on the Mac requires one to execute (once) in a Terminal

```
defaults write TeXShop OtherTeXExtensions -array-add "m"
```

- [Table A.2](#) gives the template for the `Dir/*.m` function-sections. The format for a example/test-section is similar.

- Any figures from examples should be generated and then saved for later inclusion with the following (which finally works properly for MATLAB 2017+)

```
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10]);% cm
print('-depsc2','filename')
```

If it is a suitable replacement for an existing graphic, then move it into the `Dir/Figs` folder. Include such a graphic into the LaTeX document with (do *not* postfix with `.eps` or `.pdf`)

```
\includegraphics[scale=0.9]{filename}
```

- In figures and other graphics, do *not* resize/scale fixed width constructs: instead use `\linewidth` to configure large-scale layout, `em` for small-widths, and `ex` for small-heights.
- For every function, generally include at the start of the function a simple example of its use. The example is only to be executed when the function is invoked with no input arguments (`if nargin==0`).

When appropriate, if a function is invoked with no output arguments (`if nargout==0`), then draw some reasonable graph of the results.

- In all MATLAB/Octave code, prefer camel case for variable names (not underscores).
- When a function is ‘finalised’, wrap (most) of the lines to be no more than 60 characters so that readers looking at the source can read the plain text reasonably.
- In the documentation (e.g., [Higham 1998](#), Ch. 4): write actively, not passively (e.g., avoid “-tion” words, and avoid “is/are verbed” phrases); avoid wishy-washy “can”; use the present tense; cross-reference precisely; avoid useless padding such as “note that”; and so on.

Table A.1: example `Dir/*.tex` file to typeset in the master document a function-section, say `fun.m`, and maybe the test/example-sections.

```

1  % input *.m files for ... Author, date
2  %!TEX root = ../Doc/eqnFreeDevMan.tex
3  \chapter{...}
4  \label{ch:...}
5  \localtableofcontents
6  Introduction...
7  \input{../Dir/fun.m} % prefix associated files with 'fun'
8  \input{../Dir/funExample.m}
9  ...
10 \begin{devMan}
11 \section{To do}
12 ...
13 \section{Miscellaneous tests}
14 \input{../Dir/funTest.m}
15 ...
16 \end{devMan}
```

Table A.2: template for a function-section `Dir/*.m` file.

```

1  % Short explanation for users typing "help fun"
2  % Author, date
3  %!TEX root = ../Doc/eqnFreeDevMan.tex
4  %{
5  \section{\texttt{...}: ...}
6  \label{sec:...}
7  \localtableofcontents
8  \subsection{Introduction}
9  Overview LaTeX explanation.
10 \begin{matlab}
11 %}
12 function ...
13 %
14 \end{matlab}
15 \paragraph{Input} ...
16 \paragraph{Output} ...
17 \begin{devMan}
18 Repeated as desired:
19 LaTeX between end-matlab and begin-matlab
20 \begin{matlab}
21 %}
22 Matlab code between \%} and \%{
23 %
24 \end{matlab}
25 Concluding LaTeX before following final lines.
26 \end{devMan}
27 %}
```

Appendix B Aspects of developing a ‘toolbox’ for patch dynamics

Chapter contents

B.1	Macroscale grid	288
B.2	Macroscale field variables	289
B.3	Boundary and coupling conditions	290
B.4	Mesotime communication	291
B.5	Projective integration	292
B.6	Lift to many internal modes	293
B.7	Macroscale closure	294
B.8	Exascale fault tolerance	295
B.9	Link to established packages	296

This appendix documents sketchy further thoughts on aspects of the development.

B.1 Macroscale grid

The patches are to be distributed on a macroscale grid: the j th patch ‘centred’ at position $\vec{X}_j \in \mathbb{X}$. In principle the patches could move, but let’s keep them fixed in the first version. The simplest macroscale grid will be rectangular (`meshgrid`), but we plan to allow a deformed grid to secondly cater for boundary fitting to quite general domain shapes \mathbb{X} . And plan to later allow for more general interconnect networks for more topologies in application.

B.2 Macroscale field variables

The researcher/user has to know an appropriate set of macroscale field variables $\vec{U}(t) \in \mathbb{R}^{d_U}$ for each patch. For example, first they might be a simple average over a core of a patch of all of the micro-field variables; second, they might be a subset of the average micro-field variables; and third in general the macro-variables might be a nonlinear function of the micro-field variables (such as temperature is the average speed squared). The core might be just one point, or a sizeable fraction of the patch.

The mapping from microscale variable to macroscale variables is often termed the restriction.

In practice, users may not choose an appropriate set of macro-variables, so will eventually need to code some diagnostic to indicate a failure of the assumed closure.

B.3 Boundary and coupling conditions

The physical domain boundary conditions are distinct from the conditions coupling the patches together. Start with physical boundary conditions of periodicity in the macroscale.

Second, assume the physical boundary conditions are that the macro-variables are known at macroscale grid points around the boundary. Then the issue is to adjust the interpolation to cater for the boundary presence and shape. The coupling conditions for the patches should cater for the range of Robin-like boundary conditions, from Dirichlet to Neumann. Two possibilities arise: direct imposition of the coupling action ([Roberts & Kevrekidis 2007](#)), or control by the action.

Third, assume that some of the patches have some edges coincident with the boundary of the macroscale domain \mathbb{X} , and it is on these edges that macroscale physical boundary conditions are applied. Then the interpolation from the core of these edge patches is the same as the second case of prescribed boundary macro-variables. An issue is that each boundary patch should be big enough to cater for any spatial boundary layers transitioning from the applied boundary condition to the interior slow evolution.

Alternatively, we might have the physical boundary condition constrain the interpolation between patches.

Often microscale simulations are easiest to write when ‘periodic’ in microscale space. To cater for this we should also allow a control at perhaps the quartiles of a micro-periodic simulator.

B.4 Mesotime communication

Since communication limits large scale parallelism, a first step in reducing communication will be to implement only updating the coupling conditions when necessary. Error analysis indicates that updating on times longer the microscale times and shorter than the macroscale times can be effective (Bunder et al. 2016). Implementations can communicate one or more derivatives in time, as well as macroscale variables.

At this stage we can effectively parallelise over patches: first by simply using Matlab’s `parfor`. Probably not using a GPU as we probably want to leave GPUs for the black-box to utilise within each patch.

B.5 Projective integration

Have coded several schemes.

Should not need an implicit scheme as the fast dynamics are meant to be only in the micro variables, and the slow dynamics only in the macroscale variables. However, it could be that the macroscale variables have fast oscillations and it is only the amplitude of the oscillations that are slow. Perhaps need to detect and then fix or advise, perhaps via DMD.

A further stage is to implement a projective integration scheme for stochastic macroscale variables: this is important because the averaging over a core of microscale roughness will almost invariably have at least some stochastic legacy effect. [Calderon \(2007\)](#) did some useful research on stochastic projective integration.

B.6 Lift to many internal modes

In most problems the number of macroscale variables at any given position in space, $d_{\vec{U}}$, is less than the number of microscale variables at a position, $d_{\vec{u}}$; often much less ([Kevrekidis & Samaey 2009](#), e.g.). In this case, every time we start a patch simulation we need to provide $d_{\vec{u}} - d_{\vec{U}}$ data at each position in the patch: this is lifting. The first methodology is to first guess, then run repeated short bursts with reinitialisation, until the simulation reaches a slow manifold (e.g., `cdmc()`). Then run the real simulation.

If the time taken to reach a local quasi-equilibrium is too long, then it is likely that the macroscale closure is bad and the macroscale variables need to be extended.

A second step is to cater for cases where the slow manifold is stochastic or is surrounded by fast waves: when it is hard to detect the slow manifold, or the slow manifold is not attractive.

B.7 Macroscale closure

In some circumstances a researcher/user will not code in a restriction the appropriately set of macroscale variables for a complete closure of the macroscale. For example, in thin film fluid dynamics at low Reynolds number the only macroscale variable is the fluid depth; however, at higher Reynolds number, circa ten, the inertia of the fluid becomes important and the macroscale variables must additionally include a measure of the mean lateral velocity/momentum ([Roberts & Li 2006](#), e.g.).

At some stage we need to detect any flaw in the closure implied by a restriction, and perhaps suggest additional appropriate macroscale variables, or at least their characteristics. Indeed, a poor closure and a stochastic slow manifold are really two faces of the same problem: the problem is that the chosen macroscale variables do not have a unique evolution in terms of themselves. A good resolution of the issue will account for both faces.

B.8 Exascale fault tolerance

MATLAB/Octave is probably not an appropriate vehicle to deal with real exascale faults. However, we should cater by coding procedures for fault tolerance and testing them at least synthetically. Eventually provide hooks to a user routine to be invoked under various potential scenarios. The nature of fault tolerant algorithms will vary depending upon the scenario, even assuming that each patch burst is executed on one CPU (or closely coupled CPUS): if there are many more CPUS than patches, then maybe simply duplicate all patch simulations; if many fewer CPUS than patches, then an asynchronous scheduling of patch bursts should effectively cater for recomputation of failed bursts; if comparable CPUS to patches, then more subtle action is needed.

Once mesotime communication and projective integration is provided, a recomputation approach to intermittent hardware faults should be effective because we then have the tools to restart a burst from available macroscale data. Should also explore proceeding with a lower order interpolation that misses the faulty burst—because an isolated lower order interpolation probably will not affect the global order of error (it does not in approximating some boundary conditions ([Gustafsson 1975, Svard & Nordstrom 2006](#))).

B.9 Link to established packages

Several molecular/particle/agent based codes are well developed and used by a wide community of researchers. Plan to develop hooks to use some such codes as the microscale simulators on patches. First, may connect to LAMMPS ([Plimpton et al. 2016](#)). Second, will evaluate performance, issues, and then consider what other established packages are most promising.

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.
- Biezemans, R. A., Le Bris, C., Legoll, F. & Lozinski, A. (2022), Non-intrusive implementation of multiscale finite element methods: an illustrative example, Technical report, <https://arxiv.org/abs/2204.06852>.
- 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.
- Calderon, C. P. (2007), ‘Local diffusion models for stochastic reacting systems: estimation issues in equation-free numerics’, *Molecular Simulation* **33**(9–10), 713–731.
- Cao, M. & Roberts, A. J. (2012), Modelling 3d turbulent floods based upon the smagorinski large eddy closure, in P. A. Brandner & B. W. Pearce, eds, ‘18th Australasian Fluid Mechanics Conference’.
<http://people.eng.unimelb.edu.au/imarusic/proceedings/18/70%20-%20Cao.pdf>
- Cao, M. & Roberts, A. J. (2013), Multiscale modelling couples patches of wave-like simulations, in S. McCue, T. Moroney, D. Mallet & J. Bunder, eds, ‘Proceedings of the 16th Biennial Computational Techniques and Applications Conference, CTAC-2012’, Vol. 54 of *ANZIAM J.*, pp. C153–C170.

- Cao, M. & Roberts, A. J. (2016a), ‘Modelling suspended sediment in environmental turbulent fluids’, *J. Engrg. Maths* **98**(1), 187–204.
- Cao, M. & Roberts, A. J. (2016b), ‘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. (2005a), ‘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. (2005b), ‘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. (2003a), ‘Computing in the past with forward integration’, *Phys. Lett. A* **321**, 335–343.
- Gear, C. W. & Kevrekidis, I. G. (2003b), ‘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. (2003c), ‘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>.
- Gustafsson, B. (1975), ‘The convergence rate for difference approximations to mixed initial boundary value problems’, *Mathematics of Computation* **29**(10), 396–406.

- Hereman, W. (2009), Shallow water waves and solitary waves, in ‘Mathematics of Complexity and Dynamical Systems’, Springer, New York, pp. 8112–8125.
- Higham, N. J. (1998), *Handbook of writing for the mathematical sciences*, 2nd edition edn, SIAM.
- 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 anal-

- ysis 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>].
- Plimpton, S., Thompson, A., Shan, R., Moore, S., Kohlmeyer, A., Crozier, P. & Stevens, M. (2016), Large-scale atomic/molecular massively parallel simulator, Technical report, <http://lammps.sandia.gov>.
- 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. & Li, Z. (2006), ‘An accurate and comprehensive model of thin fluid flows with inertia on curved substrates’, *J. Fluid Mech.* **553**, 33–73.
- 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.
- Svard, M. & Nordstrom, J. (2006), ‘On the order of accuracy for difference approximations of initial-boundary value problems’, *Journal of Computational Physics* **218**, 333–352.
- 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://pubs.siam.org/sam-bin/dbq/article/25797>.
- Wikipedia (2022), ‘Divided differences’.
https://en.wikipedia.org/wiki/Divided_differences

- Wilson, S. K., Hunt, R. & Duffy, B. R. (2000), 'The rate of spreading in spin coating', *J. Fluid Mech.* **413**, 65–88.