

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

A. J. Roberts<sup>\*</sup>      John Maclean<sup>†</sup>      J. E. Bunder<sup>‡</sup>

October 19, 2023

<sup>\*</sup> School of Mathematical Sciences, University of Adelaide, South Australia. <https://profajroberts.github.io>, <http://orcid.org/0000-0001-8930-1552>

<sup>†</sup> School of Mathematical Sciences, University of Adelaide, South Australia. <http://www.adelaide.edu.au/directory/john.maclean>

<sup>‡</sup> 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

---

<b>1</b>	<b>Introduction</b>	<b>8</b>
<b>2</b>	<b>Projective integration of deterministic ODEs</b>	<b>11</b>
2.1	Introduction . . . . .	12
2.2	PIRK2(): projective integration of second-order accuracy . . . . .	15
2.2.1	Introduction . . . . .	15
2.2.2	If no arguments, then execute an example . . . . .	17
2.2.3	The projective integration code . . . . .	18
2.2.4	If no output specified, then plot the simulation . . . . .	21
2.3	egPIMM: Example projective integration of Michaelis–Menton kinetics . . . . .	22
2.4	PIG(): Projective Integration via a General macroscale integrator	26
2.4.1	Introduction . . . . .	26
2.4.2	If no arguments, then execute an example . . . . .	28
2.4.3	The projective integration code . . . . .	29
2.4.4	If no output specified, then plot the simulation . . . . .	32
2.5	PIRK4(): projective integration of fourth-order accuracy . . . . .	33
2.5.1	Introduction . . . . .	33
2.5.2	The projective integration code . . . . .	35
2.5.3	If no output specified, then plot the simulation . . . . .	39
2.6	cdmc(): constraint defined manifold computing . . . . .	40
2.7	Example: PI using Runge–Kutta macrosolvers . . . . .	41
2.8	Example: Projective Integration using General macrosolvers	44
2.9	Explore: Projective Integration using constraint-defined manifold computing . . . . .	46
2.10	To do/discuss . . . . .	48

<b>3 Patch scheme for given microscale discrete space system</b>	<b>49</b>
3.1 <code>configPatches1()</code> : configure spatial patches in 1D . . . . .	51
3.1.1 If no arguments, then execute an example . . . . .	54
3.1.2 Parse input arguments and defaults . . . . .	56
3.1.3 The code to make patches and interpolation . . . . .	57
3.1.4 Set ensemble inter-patch communication . . . . .	60
3.2 <code>patchSys1()</code> : interface 1D space to time integrators . . . . .	63
3.3 <code>patchEdgeInt1()</code> : sets patch-edge values from interpolation over the 1D macroscale . . . . .	65
3.3.1 Interpolate in the x-direction . . . . .	67
3.4 <code>homogenisationExample</code> : simulate heterogeneous diffusion in 1D . . . . .	68
3.4.1 Script to simulate via stiff or projective integration . . . . .	68
3.4.2 <code>heteroDiff()</code> : heterogeneous diffusion . . . . .	71
3.4.3 <code>heteroBurst()</code> : a burst of heterogeneous diffusion . . . . .	72
3.5 <code>homoDiffEdgy1</code> : computational homogenisation of a 1D heterogeneous diffusion by simulation on small patches . . . . .	73
3.5.1 Script code to simulate heterogeneous diffusion systems . . . . .	73
3.5.2 <code>heteroDiff()</code> : heterogeneous diffusion . . . . .	77
3.6 <code>BurgersExample</code> : simulate Burgers' PDE on patches . . . . .	79
3.6.1 Script code to simulate a microscale space-time map . . . . .	79
3.6.2 Alternatively use projective integration . . . . .	80
3.6.3 <code>burgersMap()</code> : discretise the PDE microscale . . . . .	81
3.6.4 <code>burgerBurst()</code> : code a burst of the patch map . . . . .	82
3.7 <code>waterWaveExample</code> : simulate a water wave PDE on patches . . . . .	83
3.7.1 Script code to simulate wave systems . . . . .	84
3.7.2 <code>idealWavePDE()</code> : ideal wave PDE . . . . .	86
3.7.3 <code>waterWavePDE()</code> : water wave PDE . . . . .	87
3.8 <code>homoWaveEdgy1</code> : computational homogenisation of a 1D wave by simulation on small patches . . . . .	89
3.8.1 Script code to simulate heterogeneous wave systems . . . . .	89
3.8.2 <code>heteroWave()</code> : wave in heterogeneous media with weak viscous damping . . . . .	92
3.9 <code>waveEdgy1</code> : simulate a 1D, first-order, wave PDE on small patches . . . . .	94

3.9.1	Script code to simulate heterogeneous wave systems . . . . .	94
3.9.2	waveFirst(): first-order wave PDE . . . . .	98
3.10	Eckhardt2210eg2: example of a 1D heterogeneous diffusion by simulation on small patches . . . . .	99
3.10.1	Simulate heterogeneous diffusion systems . . . . .	100
3.10.2	heteroDiffF(): forced heterogeneous diffusion . . . . .	102
3.11	EckhardtEquilib: find an equilibrium of a 1D heterogeneous diffusion via small patches . . . . .	103
3.12	EckhardtEquilibErrs: explore errors in equilibria of a 1D heterogeneous diffusion on small patches . . . . .	105
3.13	Eckhardt2210eg1: example of 1D space-time heterogeneous diffusion via computational homogenisation with projective integration and small patches . . . . .	110
3.13.1	Simulate heterogeneous diffusion systems . . . . .	111
3.13.2	heteroBurstF(): a burst of heterogeneous diffusion . .	113
3.14	homoLanLif1D: computational homogenisation of a 1D heterogeneous Landau–Lifshitz by simulation on small patches . . .	114
3.14.1	Script code to simulate heterogeneous diffusion systems	115
3.14.2	Spectrum of the coded patch system . . . . .	117
3.14.3	heteroLanLif1D(): heterogeneous Landau–Lifshitz PDE	119
3.15	Combescure2022: simulation and continuation of a 1D example nonlinear elasticity, via patches . . . . .	120
3.15.1	Configure heterogeneous toy elasticity systems . . . . .	122
3.15.2	Simulate in time . . . . .	124
3.15.3	MatCont continuation . . . . .	125
3.15.4	matContSys: basic function for MatCont analysis . .	127
3.15.5	dudtSys(): wraps around the patch wrapper . . . . .	127
3.15.6	heteroNLE(): forced heterogeneous elasticity . . . . .	128
3.16	hyperDiffHetero: simulate a heterogeneous hyper-diffusion PDE in 1D on patches . . . . .	130
3.16.1	Heterogeneous hyper-diffusion PDE inside patches .	132
3.17	SwiftHohenbergPattern: patterns of the Swift–Hohenberg PDE in 1D on patches . . . . .	133
3.17.1	The Swift–Hohenberg PDE and BCs inside patches .	135
3.17.2	theRes(): wrapper function to zero for equilibria . .	136

3.18	<code>SwiftHohenbergHetero</code> : patterns of a heterogeneous Swift–Hohenberg PDE in 1D on patches . . . . .	137
3.18.1	Heterogeneous SwiftHohenberg PDE+BCs inside patches	143
3.18.2	<code>theRes()</code> : a wrapper function . . . . .	144
3.19	<code>theRes()</code> : wrapper function to zero for equilibria . . . . .	145
3.20	<code>quasiLogAxes()</code> : transforms current axes of plot(s) to quasi-log	146
3.20.1	<code>tickingQuasiLogAxes()</code> : typeset ticks and labels on an axis . . . . .	149
3.21	<code>patchEdgeIntCore()</code> : sets multi-D patch face values from 1D macroscale interpolation in one direction . . . . .	151
3.21.1	Periodic macroscale interpolation schemes . . . . .	152
3.21.2	Non-periodic macroscale interpolation . . . . .	154
3.21.3	<code>patchEdgeInt1test</code> : test the 1D patch coupling . . . . .	156
<b>4</b>	<b>Patches in 2D space</b>	<b>163</b>
4.1	<code>configPatches2()</code> : configures spatial patches in 2D . . . . .	164
4.1.1	If no arguments, then execute an example . . . . .	167
4.1.2	Parse input arguments and defaults . . . . .	170
4.1.3	The code to make patches . . . . .	172
4.1.4	Set ensemble inter-patch communication . . . . .	175
4.2	<code>patchSys2()</code> : interface 2D space to time integrators . . . . .	179
4.3	<code>patchEdgeInt2()</code> : sets 2D patch face values from 2D macroscale interpolation . . . . .	181
4.3.1	Interpolate over the two successive directions . . . . .	183
4.4	<code>wave2D</code> : example of a wave on patches in 2D . . . . .	184
4.4.1	Check on the linear stability of the wave PDE . . . . .	184
4.4.2	Execute a simulation . . . . .	185
4.4.3	<code>wavePDE()</code> : Example of simple wave PDE inside patches	186
4.5	<code>homoDiffEdgy2</code> : computational homogenisation of a 2D diffusion via simulation on small patches . . . . .	188
4.5.1	Compute Jacobian and its spectrum . . . . .	189
4.5.2	<code>heteroDiff2()</code> : heterogeneous diffusion . . . . .	191
4.6	<code>homoDiffSoln2</code> : steady state of a 2D heterogeneous diffusion via small patches . . . . .	192
4.6.1	Microscale discretisation inside patches . . . . .	194

4.6.2	Function to explore errors in the patch scheme . . . . .	195
4.7	<code>monoscaleDiffEquil2</code> : equilibrium of a 2D monoscale heterogeneous diffusion via small patches . . . . .	198
4.7.1	<code>monoscaleDiffForce2()</code> : microscale discretisation inside patches of forced diffusion PDE . . . . .	200
4.8	<code>twoscaleDiffEquil2</code> : equilibrium of a 2D twoscale heterogeneous diffusion via small patches . . . . .	201
4.8.1	<code>twoscaleDiffForce2()</code> : microscale discretisation inside patches of forced diffusion PDE . . . . .	203
4.9	<code>twoscaleDiffEquil2Errs</code> : errors in equilibria of a 2D twoscale heterogeneous diffusion via small patches . . . . .	204
4.9.1	<code>twoscaleDiffForce2()</code> : microscale discretisation inside patches of forced diffusion PDE . . . . .	209
4.10	<code>abdulleDiffEquil2</code> : equilibrium of a 2D multiscale heterogeneous diffusion via small patches . . . . .	211
4.10.1	<code>abdulleDiffForce2()</code> : microscale discretisation inside patches of forced diffusion PDE . . . . .	213
4.11	<code>randAdvecDiffEquil2</code> : equilibrium of a 2D random heterogeneous advection-diffusion via small patches . . . . .	214
4.11.1	<code>randAdvecDiffForce2()</code> : microscale discretisation inside patches of forced diffusion PDE . . . . .	216
4.12	<code>homoWaveEdgy2</code> : computational homogenisation of a forced, non-autonomous, 2D wave via simulation on small patches . .	218
4.12.1	<code>heteroWave2()</code> : heterogeneous Waves . . . . .	221
4.13	<code>SwiftHohenberg2dPattern</code> : patterns of the Swift–Hohenberg PDE in 2D on patches . . . . .	223
4.13.1	The Swift–Hohenberg PDE and BCs inside patches . .	227
4.13.2	<code>patchEdgeInt2test</code> : tests 2D patch coupling . . . . .	228
<b>5</b>	<b>Patches in 3D space</b>	<b>233</b>
5.1	<code>configPatches3()</code> : configures spatial patches in 3D . . . . .	234
5.1.1	If no arguments, then execute an example . . . . .	238
5.1.2	<code>heteroWave3()</code> : heterogeneous Waves . . . . .	241
5.1.3	Parse input arguments and defaults . . . . .	241
5.1.4	The code to make patches . . . . .	244
5.1.5	Set ensemble inter-patch communication . . . . .	247
5.2	<code>patchSys3()</code> : interface 3D space to time integrators . . . . .	250

5.3	<code>patchEdgeInt3()</code> : sets 3D patch face values from 3D macroscale interpolation . . . . .	252
5.3.1	Interpolate over the three successive directions . . . . .	254
5.4	<code>homoDiffEdgy3</code> : computational homogenisation of a 3D diffusion via simulation on small patches . . . . .	256
5.4.1	Simulate heterogeneous diffusion . . . . .	256
5.4.2	Compute Jacobian and its spectrum . . . . .	258
5.4.3	<code>heteroDiff3()</code> : heterogeneous diffusion . . . . .	260
5.5	<code>homoDiffBdryEqui13</code> : equilibrium via computational homogenisation of a 3D heterogeneous diffusion on small patches . . . . .	262
5.5.1	<code>microDiffBdry3()</code> : 3D forced heterogeneous diffusion with boundaries . . . . .	264
5.6	<code>heteroDispersiveWave3</code> : heterogeneous Dispersive Waves from 4th order PDE . . . . .	266
5.6.1	<code>heteroDispWave3()</code> : PDE function of 4th-order heterogeneous dispersive waves . . . . .	268
5.6.2	<code>patchEdgeInt3test</code> : tests 3D patch coupling . . . . .	269
5.7	To do for patches . . . . .	274
<b>6</b>	<b>Matlab parallel computation of the patch scheme</b> . . . . .	<b>275</b>
6.1	<code>chanDispSpmd</code> : simulation of a 1D shear dispersion via simulation on small patches across a channel . . . . .	276
6.1.1	Simulate heterogeneous advection-diffusion . . . . .	277
6.1.2	Plot the solution . . . . .	279
6.1.3	<code>microBurst</code> function for Projective Integration . . . . .	281
6.1.4	<code>chanDispMicro()</code> : heterogeneous 2D advection-diffusion in a long thin channel . . . . .	282
6.2	<code>rotFilmSpmd</code> : simulation of a 2D shallow water flow on a rotating heterogeneous substrate . . . . .	283
6.2.1	Simulate heterogeneous advection-diffusion . . . . .	284
6.2.2	Plot the solution . . . . .	286
6.2.3	<code>microBurst</code> function for Projective Integration . . . . .	288
6.2.4	<code>rotFilmMicro()</code> : 2D shallow water flow on a rotating heterogeneous substrate . . . . .	289
6.3	<code>homoDiff31spmd</code> : computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion . . . . .	291

6.3.1	Simulate heterogeneous diffusion . . . . .	292
6.3.2	Plot the solution . . . . .	294
6.3.3	microBurst function for Projective Integration . . . . .	295
6.4	RK2mesoPatch() . . . . .	296
6.5	To do . . . . .	300
<b>A</b>	<b>Create, document and test algorithms</b>	<b>301</b>
<b>B</b>	<b>Aspects of developing a ‘toolbox’ for patch dynamics</b>	<b>304</b>
B.1	Macroscale grid . . . . .	305
B.2	Macroscale field variables . . . . .	306
B.3	Boundary and coupling conditions . . . . .	307
B.4	Mesotime communication . . . . .	308
B.5	Projective integration . . . . .	309
B.6	Lift to many internal modes . . . . .	310
B.7	Macroscale closure . . . . .	311
B.8	Exascale fault tolerance . . . . .	312
B.9	Link to established packages . . . . .	313

---

## 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 folders `Patch` and `ProjInt` of this toolbox in a path searched by MATLAB/Octave (but *not* the folder `SandpitPlay` as that contains experimental development functions). 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.
- In MATLAB, the axis labelling works best if one executes  
`set(groot,'defaultTextInterpreter','latex')`

**Verification** Most of these schemes have analytically 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 given 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, and [Divahar et al. \(2022\)](#), [Bunder et al. \(2021\)](#) established excellent 2D staggered patch schemes for waves.
- A heterogeneous microscale is more difficult, but [Bunder et al. \(2020\)](#), [Bunder & Roberts \(2022\)](#) developed a new ‘edgy’ inter-patch interpolation that is analytically proven to be excellent for simulating the macroscale homogenised dynamics of microscale heterogeneous systems in multiple space dimensions—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 powerful and flexible MATLAB/Octave toolbox of equation-free algorithms. Initially the algorithms are basic, and the ongoing program is developing 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 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

---

## 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.

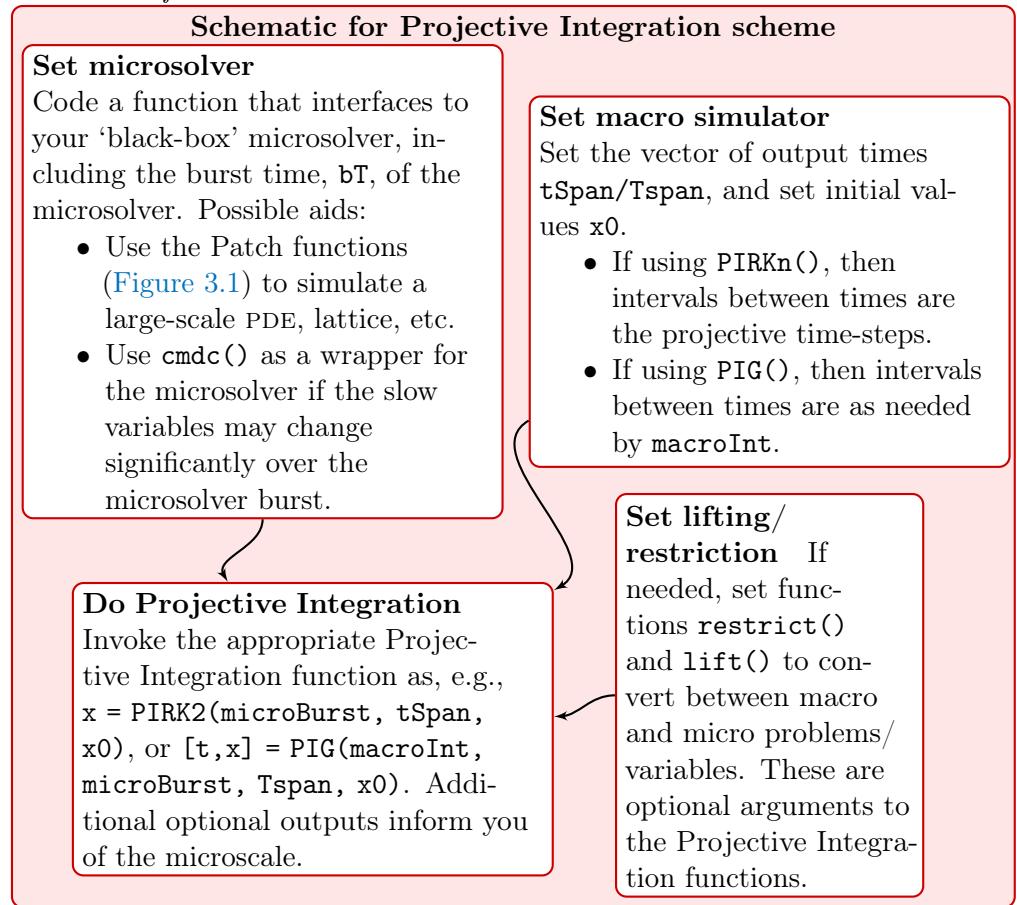
The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

[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.

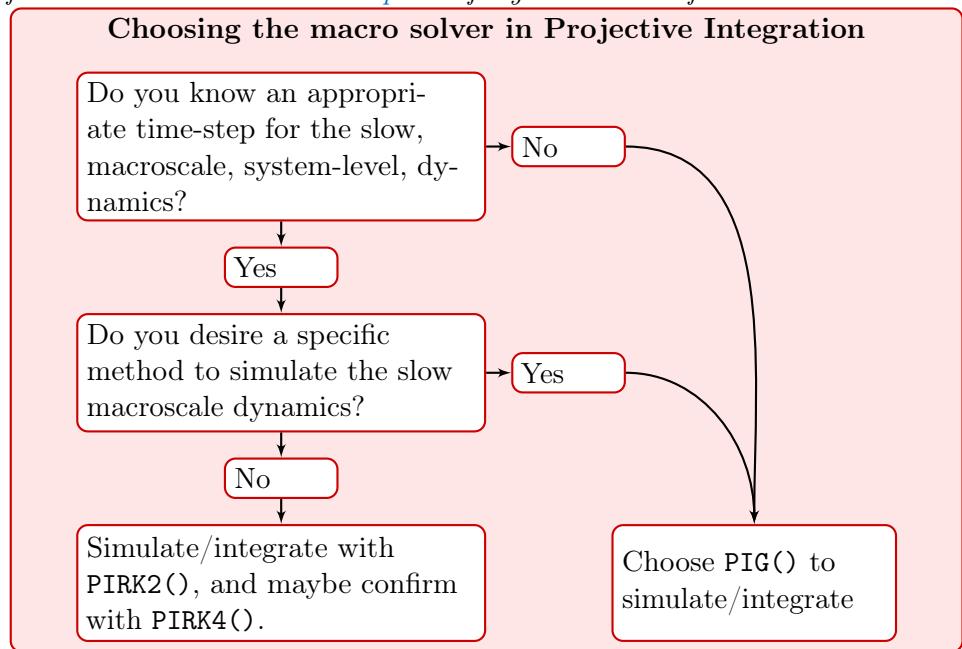
*Figure 2.1: The Projective Integration method greatly accelerates simulation/integration of a system exhibiting multiple time scales. The Projective Integration Chapter 2 presents several separate functions, as well as several optional wrapper functions that may be invoked. This chart overviews constructing 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

### 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)
```

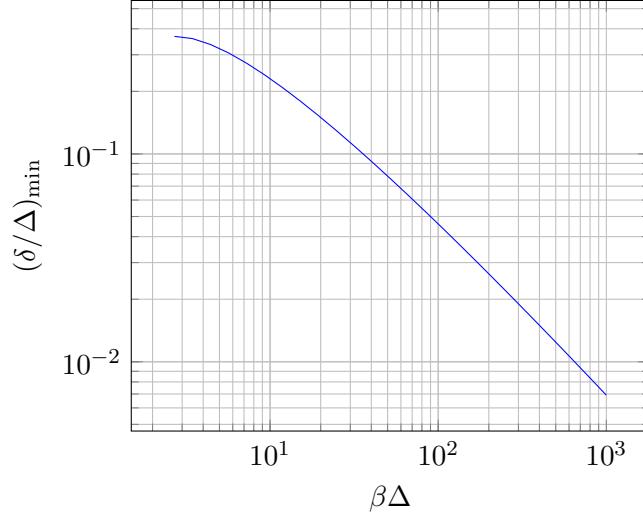
- Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row  $n$ -vector of the starting state; `bT`, *optional*, the total time to simulate in the burst—if your `microBurst()` determines the burst time, then replace `bT` in the argument list by `varargin`.
- Outputs: `tOut`, the column 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  $\ell$ -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.
- `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
```

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



**Choose a long enough burst length** Suppose: firstly, you have some desired relative accuracy  $\varepsilon$  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  $\alpha$ ; and thirdly, the rate of *decay* of your fast modes are faster than the lower bound  $\beta$  (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,  $\Delta = \text{diff}(\text{tSpan})$ , 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  $\mathbf{x}$  versus  $\text{tSpan}$ .

- $\mathbf{x}$ , an  $\ell \times n$  array of the approximate solution vector. Each row is an estimated state at the corresponding time in  $\text{tSpan}$ . The simplest usage is then  $\mathbf{x} = \text{PIRK2}(\text{microBurst}, \text{tSpan}, \mathbf{x}_0, bT)$ .

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

- $\mathbf{tms}$ , optional, is an  $L$  dimensional column vector containing the microscale times within the burst simulations, each burst separated by NaN;
- $\mathbf{xms}$ , 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  $\mathbf{tms}$  and helps visualise details of the solution.
- $\mathbf{rm}$ , 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  $2\ell$  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–Menton enzyme kinetics** Integrate a burst of length `bT` of the ODES for the Michaelis–Menton enzyme kinetics at parameter  $\epsilon$  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

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

```

### 2.2.3 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,:);

```

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

```

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  $\epsilon$ .

**Invoke projective integration** Clear, and set the scale separation parameter  $\epsilon$  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  $2\epsilon$ , 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.

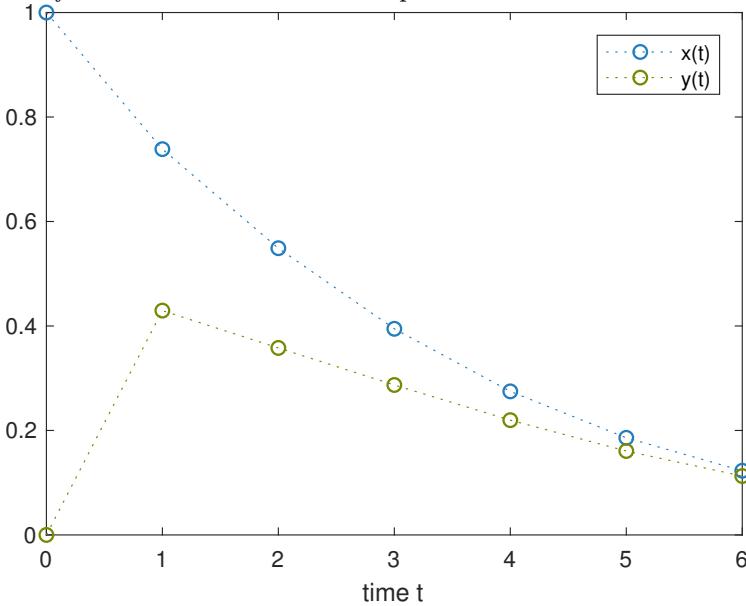


Figure 2.5 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable  $x(t)$  is also affected by the initial transient (hence other schemes which ‘freeze’ slow variables are less accurate).

**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  $3\epsilon$ .

```

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  $\epsilon$  inherited from above. Code ODES in function  $dMMdt$  with variables  $x = x(1)$  and  $y = x(2)$ . Starting at time  $ti$ , and state  $xi$  (row), we here simply use MATLAB/Octave’s `ode23/lsode` to integrate a burst in time.

```

15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [-x(1)+(x(1)+0.5)*x(2)
18                      1/MMepsilon*( x(1)-(x(1)+1)*x(2) )];

```

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.

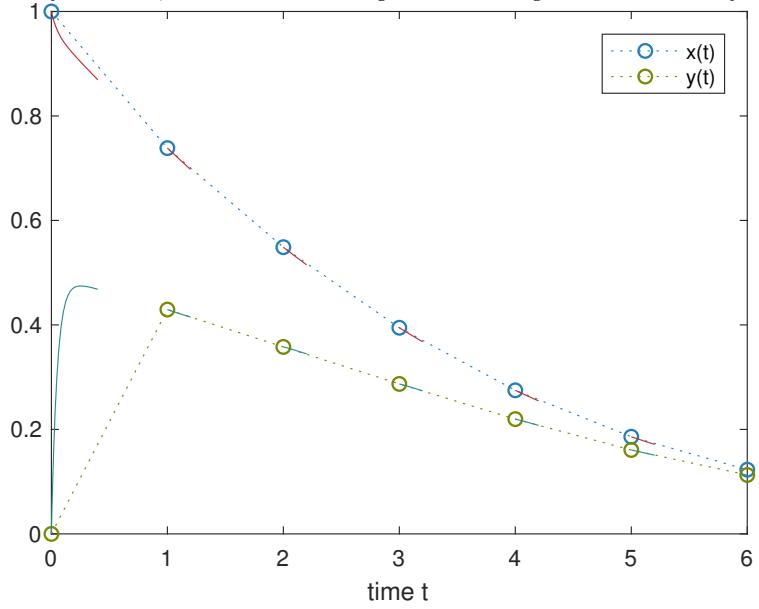
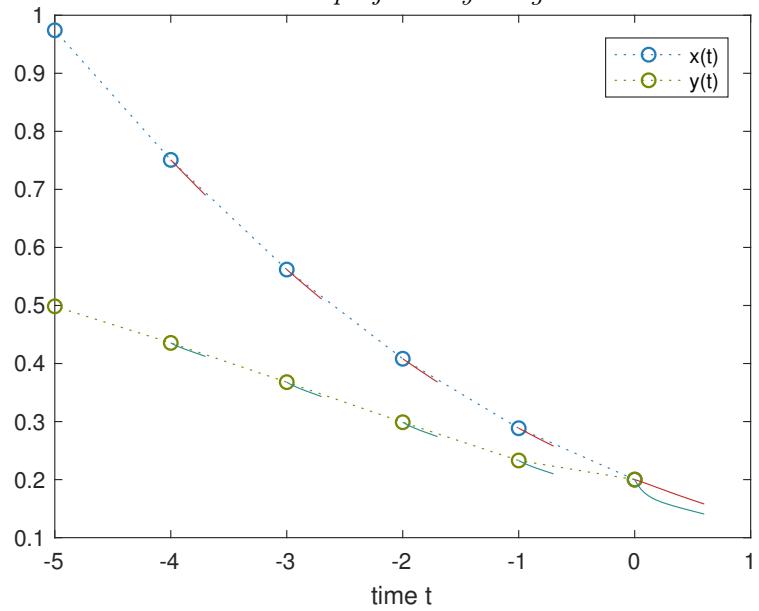


Figure 2.6: Michaelis–Menten enzyme kinetics at  $\epsilon = 0.1$  simulated backward with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to projectively integrate backward in time.



```
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

### 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

$$[tbs, xbs] = \text{microBurst}(tb0, xb0)$$

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

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

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

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

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

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

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

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

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

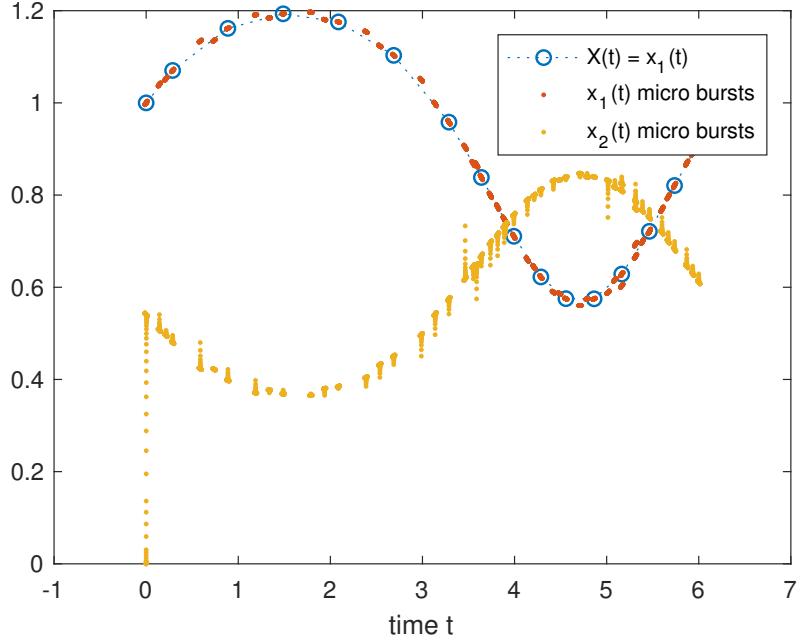
**Output** Between zero and five outputs may be requested. If there are no output arguments specified, then a plot is drawn of the computed solution **X** versus **T**. Most often you would store the first two output results of `PIG()`, via say `[T,X] = PIG(...)`.

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

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

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

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.



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

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

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

#### 2.4.2 If no arguments, then execute an example

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

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

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

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.

```
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 = nargout();
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(cdmCFlag) ' 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

### 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 % Plotting code continues...
60
61 % Upon finishing execution of the example, exit this function.
62
63 return
64 end%if no arguments
```

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

```
15 function [ts, xs] = MMburst(ti, xi, bT)
16     global MMepsilon
17     dMMdt = @(t,x) [-x(1)+(x(1)+0.5)*x(2)
18                     1/MMepsilon*( x(1)-(x(1)+1)*x(2) )];
19     if ~exist('OCTAVE_VERSION','builtin')
```

```

20      [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
21      else % octave version
22      [ts, xs] = odeOct(dMMdt, [ti ti+bT], xi);
23      end
24  end

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

```

**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  $\ell$ -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`.

- $\mathbf{x}$ , an  $\ell \times n$  array of the approximate solution vector. Each row is an estimated state at the corresponding time in  $\mathbf{tSpan}$ . The simplest usage is then  $\mathbf{x} = \text{PIRK4}(\text{microBurst}, \mathbf{tSpan}, \mathbf{x0}, \mathbf{bT})$ .

However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK4()` provides up to four optional outputs of the microscale bursts.

- $\mathbf{tms}$ , optional, is an  $L$  dimensional column vector containing the microscale times within the burst simulations, each burst separated by NaN;
- $\mathbf{xms}$ , 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  $\mathbf{tms}$  and helps visualise details of the solution.
- $\mathbf{rm}$ , optional, a struct containing the ‘remaining’ applications of the microBurst required by the Projective Integration method during the calculation of the macrostep:
  - $\mathbf{rm.t}$  is a column vector of microscale times; and
  - $\mathbf{rm.x}$  is the array of corresponding burst states.

The states  $\mathbf{rm.x}$  do not have the same physical interpretation as those in  $\mathbf{xms}$ ; the  $\mathbf{rm.x}$  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.

- $\mathbf{svf}$ , optional, a struct containing the Projective Integration estimates of the slow vector field.
  - $\mathbf{svf.t}$  is a  $4\ell$  dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along microBurst data to form a macrostep.
  - $\mathbf{svf.dx}$  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
453 end
```

This concludes PIRK4().

## 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);
```

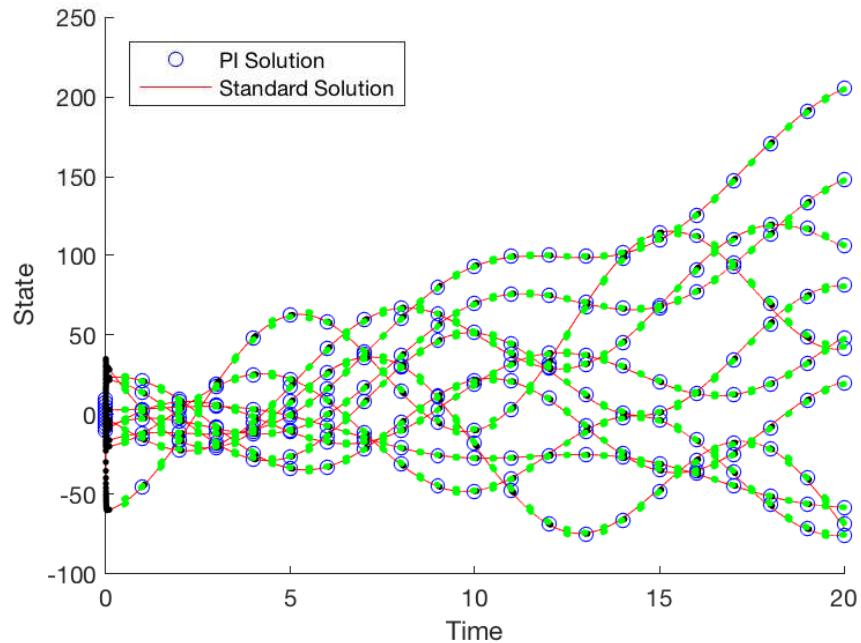
To verify, we also compute the trajectories using a standard integrator.

```
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];
```

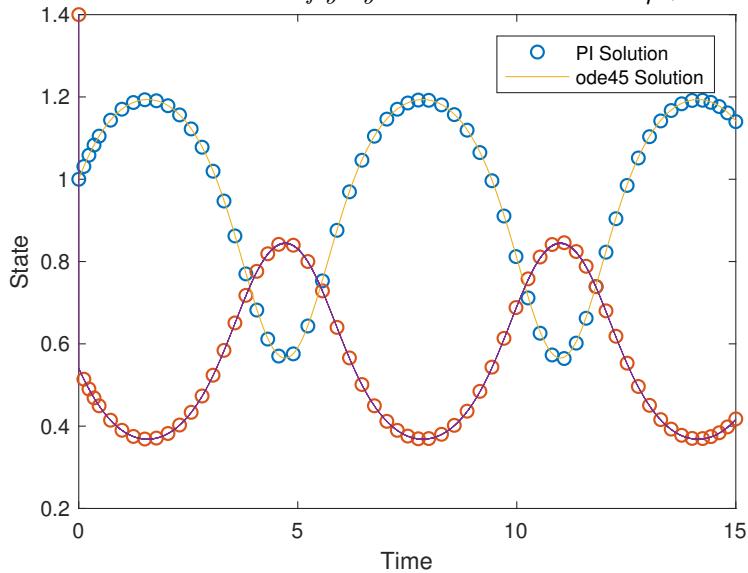
Now time and integrate the above system over `tSpan` using `PIG()` and, for comparison, a brute force implementation of `ode45/lsode`. Report the time taken by each method (in seconds).

```
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  $\epsilon$ , whereas that of `ode45()` is proportional to  $1/\epsilon$ .

If the problem is ‘semi-stiff’ (larger  $\epsilon$ ), 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.

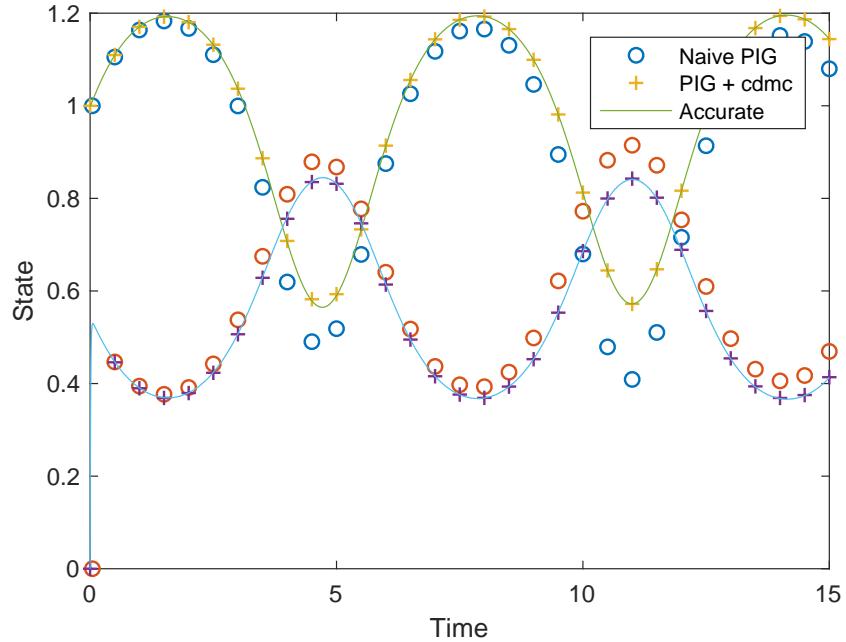
```
62 if ~exist('OCTAVE_VERSION','builtin')
63     macInt='ode45'; else macInt='odeOct'; end
64 [nt,nx] = PIG(macInt,microBurst,tSpan,x0,[],[],'no cdmc');
65 [ct,cx] = PIG(macInt,microBurst,tSpan,x0);
66 [tClassic,xClassic] = feval(macInt,dxdt,tSpan,x0);
```

[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*epsilon1` 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.

---

<sup>1</sup>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<sub>n</sub>() 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

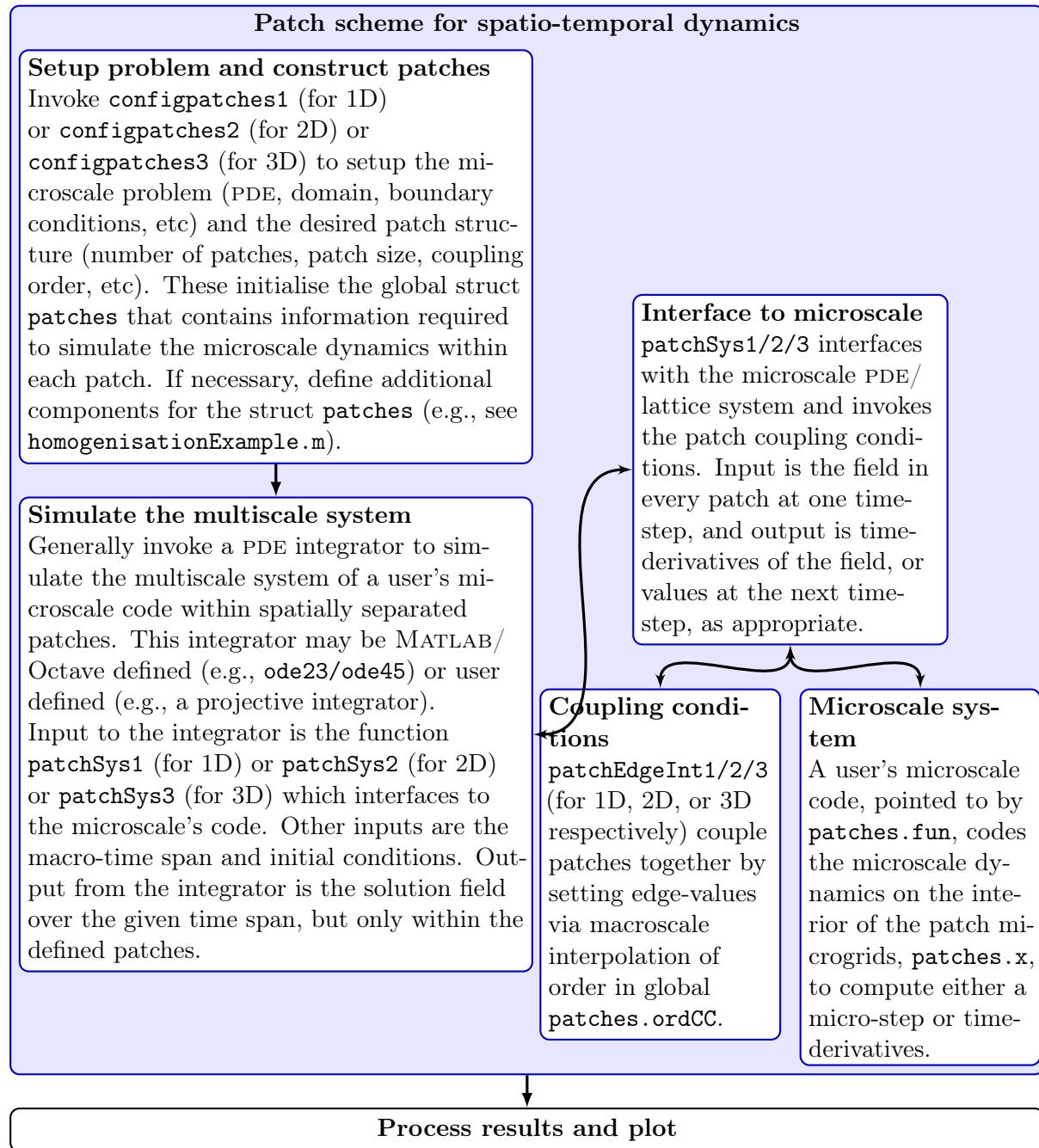
---

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(): configure spatial patches in 1D

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

```
19 function patches = configPatches1(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-10-12';
```

**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)]`.

But, if `nPatch` = 1, then the domain is the mean of `Xlim` plus/minus half the patch-width.

- `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 array, 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`, char-array, of either '`periodic`' (the default), '`equispace`', '`chebyshev`', '`usergiven`'. For all cases except '`periodic`', users *must* code into `fun` the micro-grid boundary conditions that apply at the left(right) edge of the leftmost(rightmost) patches.
  - `.bcOffset`, optional one or two element array, in the cases of '`equispace`' or '`chebyshev`' the patches are placed so the left/right macroscale boundaries are aligned to the left/right edges of the corresponding extreme patches, but offset by `bcOffset` of the sub-patch micro-grid spacing. For example, use `bcOffset=0` when applying Dirichlet boundary values on the extreme edge micro-grid points, whereas use `bcOffset=0.5` when applying Neumann boundary conditions halfway between the extreme edge micro-grid points.
  - `.X`, optional array, in the case '`usergiven`' it specifies the locations of the centres of the `nPatch` patches—the user is responsible it makes sense.
- `nPatch` is the number of spatial patches: an integer  $\geq 1$ .
- `ordCC`, must be  $\geq -1$ , is the ‘order’ of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch

coupling: where `ordCC` of 0 or  $-1$  gives spectral interpolation; and `ordCC` being odd specifies staggered spatial grids.

- `dx` (real) is usually the sub-patch micro-grid spacing in  $x$ .

However, if `Dom` is `NaN` (as for pre-2023), then `dx` actually is `ratio`, namely the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the equi-spacing of the patch mid-points—adjusted a little when `nEdge`  $> 1$ . So either `ratio` =  $\frac{1}{2}$  means the patches abut and `ratio` = 1 is overlapping patches as in holistic discretisation, or `ratio` = 1 means the patches abut. Small `ratio` should greatly reduce computational time.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch. If not using `EdgyInt`, then `nSubP/nEdge` must be odd integer so that there is/are centre-patch lattice point(s). So for the defaults of `nEdge` = 1 and not `EdgyInt`, then `nSubP` must be odd.
- 'nEdge', *optional*, default=1, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `hetCoeffs`, *optional*, default empty. Supply a 1D or 2D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size  $m_x \times n_c$ , where  $n_c$  is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the  $n_c$  coefficients being  $n_c$  parameters in some macroscale formula.
  - If `nEnsem` = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of  $(nSubP-nEdge)/2$  for not `EdgyInt`.
  - If `nEnsem`  $> 1$  (value immaterial), then reset `nEnsem := m_x` and construct an ensemble of all  $m_x$  phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry.
- `nCore`, *optional-experimental*, default one, but if more, and only for non-`EdgyInt`, then interpolates from an average over the core of a patch, a core of size `??`. Then edge values are set according to interpolation of the averages`??` or so that average at edges is the interpolant`??`

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

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

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

```
190 if nargout==0, global patches, end
191 patches.version = version;
```

- `.fun` is the name of the user’s function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)`, that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.periodic`: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- `.Cwtsr` and `.Cwtsl`, only for macro-periodic conditions, are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:microscale 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.
- `.le`, `.ri` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either

---

<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.

- [] 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

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

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

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

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

```
285 global patches
286 patches = configPatches1(@BurgersPDE, [0 2*pi], ...
287     'periodic', 8, 0, 0.06, 7);
```

Set some initial condition, with some microscale randomness.

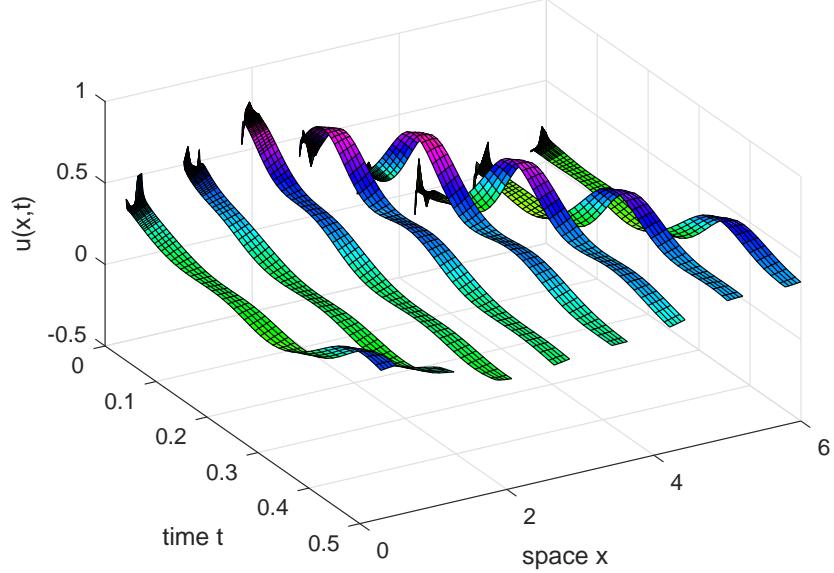
```
293 u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
```

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

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

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

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



```

317 figure(1),clf
318 if 1, patches.x([1 end],:,:, :)=nan; us=us.';
319 else us=reshape(patchEdgyInt1(us.'),[],length(ts));
320 end
321 mesh(ts,patches.x(:,us)
322 view(60,40), colormap(0.7* hsv)
323 title('Burgers PDE: patches in space, continuous time')
324 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.

```

338 if OurCf2eps(mfilename)
339 return
340 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] = odeOcts(dxdt,tSpan,x0)
11     if length(tSpan)>2, ts = tSpan;
12     else ts = linspace(tSpan(1),tSpan(end),21)';
13     end
14     lsode_options('integration method','non-stiff');
15     xs = lsode(@(x,t) dxdt(t,x),x0,ts);
16 end

```

### 3.1.2 Parse input arguments and defaults

```

355 p = inputParser;
356 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
357 addRequired(p,'fun',fnValidation);
358 addRequired(p,'Xlim',@isnumeric);
359 %addRequired(p,'Dom'); % nothing yet decided
360 addRequired(p,'nPatch',@isnumeric);
361 addRequired(p,'ordCC',@isnumeric);
362 addRequired(p,'dx',@isnumeric);
363 addRequired(p,'nSubP',@isnumeric);
364 addParameter(p,'nEdge',1,@isnumeric);
365 addParameter(p,'EdgyInt',false,@islogical);
366 addParameter(p,'nEnsem',1,@isnumeric);
367 addParameter(p,'hetCoeffs',[],@isnumeric);
368 addParameter(p,'parallel',false,@islogical);
369 addParameter(p,'nCore',1,@isnumeric);
370 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

376 patches.nEdge = p.Results.nEdge;
377 patches.EdgyInt = p.Results.EdgyInt;
378 patches.nEnsem = p.Results.nEnsem;
379 cs = p.Results.hetCoeffs;
380 patches.parallel = p.Results.parallel;
381 patches.nCore = p.Results.nCore;

```

Check parameters.

```

388 assert((nPatch==round(nPatch))&(nPatch>0) ...
389     , 'nPatch must be integer and at least one')
390 assert(Xlim(1)<Xlim(2) ...
391     , 'two entries of Xlim must be ordered increasing')
392 assert((mod(ordCC,2)==0)|(patches.nEdge==1) ...
393     , 'Cannot yet have nEdge>1 and staggered patch grids')
394 assert(3*patches.nEdge<=nSubP ...
395     , 'too many edge values requested')
396 assert(rem(nSubP,patches.nEdge)==0 ...
397     , 'nSubP must be integer multiple of nEdge')
398 if ~patches.EdgyInt, assert(rem(nSubP/patches.nEdge,2)==1 ...
399     , 'for non-edgyInt, nSubP/nEdge must be odd integer')
400 end

```

```

401 if (patches.nEnsem>1)&&(patches.nEdge>1)
402     warning('not yet tested when both nEnsem and nEdge non-one')
403 end
404 if patches.nCore>1
405     error('nCore>1 not yet available in this version')
406 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.

```

416 if ~isstruct(Dom), pre2023=isnan(Dom);
417 else pre2023=false; end
418 if pre2023, ratio=dx; dx=nan; end

```

Default macroscale conditions are periodic with evenly spaced patches.

```

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

```

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

```

435 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.

```

443 patches.periodic=false;
444 switch Dom.type
445 case {'periodic','periodic'}
446     patches.periodic=true;
447     if isfield(Dom,'bcOffset')
448         warning('bcOffset not available for Dom.type = periodic'), end
449     if isfield(Dom,'X')
450         warning('X not available for Dom.type = periodic'), end
451     case {'equispace','chebyshev'}
452         if ~isfield(Dom,'bcOffset'), Dom.bcOffset=[0;0]; end
453         if length(Dom.bcOffset)==1
454             Dom.bcOffset=repmat(Dom.bcOffset,2,1); end
455         if isfield(Dom,'X')
456             warning('X not available for Dom.type = equispace or chebyshev')
457         end
458     case 'usergiven'
459         if isfield(Dom,'bcOffset')
460             warning('bcOffset not available for usergiven Dom.type'), end
461             assert(isfield(Dom,'X'),'X required for Dom.type = usergiven')
462     otherwise
463         error([Dom.type ' is unknown Dom.type'])
464 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.

---

```
476 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  $-1$ .

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

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

```
493 patches.stag=mod(ordCC,2);
494 ordCC=ordCC+patches.stag;
495 patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

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

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

```
512 switch Dom.type
```

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

```
520 case {'periodic','periodic'}
521 X=linspace(Xlim(1),Xlim(2),nPatch+1);
522 DX=X(2)-X(1);
523 X=X(1:nPatch)+diff(X)/2;
524 pEI=patches.EdgyInt;% abbreviation
525 pnE=patches.nEdge; % abbreviation
526 if pre2023, dx = ratio*DX/(nSubP-pnE*(1+pEI))*(2-pEI);
527 else ratio = dx/DX*(nSubP-pnE*(1+pEI))/(2-pEI); end
528 patches.ratio=ratio;
```

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

```
536 if ordCC>0
537 [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
538 patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
539 end
```

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

```
549 case 'equispace'
550 halfWidth=dx*(nSubP-1)/2;
551 if nPatch>1
552 X=linspace(Xlim(1)+halfWidth-Dom.bcOffset(1)*dx ...
```

---

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

```

553             ,Xlim(2)-halfWidth+Dom.bcOffset(2)*dx ,nPatch);
554             DX=diff(X(1:2));
555         else% nPatch==1
556             warning('nPatch==1 so placing one patch at the mean of Xlim')
557             X=mean(Xlim);
558             DX=Inf;
559         end
560         width=(1+patches.EdgyInt)/2*(nSubP-1-patches.EdgyInt)*dx;
561         if DX<width*0.999999
562             warning('too many equispace patches (double overlapping)')
563         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 extreme edges aligned with the spatial domain boundaries, modified by the offset, and modified by possible ‘boundary layers’.<sup>3</sup>

```

580     case 'chebyshev'
581         halfWidth=dx*(nSubP-1)/2;
582         if nPatch>1
583             X1 = Xlim(1)+halfWidth-Dom.bcOffset(1)*dx;
584             X2 = Xlim(2)-halfWidth+Dom.bcOffset(2)*dx;
585 %             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`. We need to find `b`, the number of patches ‘glued’ together at the boundaries.

```

595     pEI=patches.EdgyInt;% abbreviation
596     pnE=patches.nEdge; % abbreviation
597     width=(1+pEI)/2*(nSubP-pnE-pEI*pnE)*dx;
598     for b=0:2:nPatch-2
599         DXmin=(X2-X1-b*width)/2*( 1-cos(pi/(nPatch-b-1)) );
600         if DXmin>width, break, end
601     end%for
602     if DXmin<width*0.999999
603         warning('too many Chebyshev patches (mid-domain overlap)')
604     end

```

Assign the centre-patch coordinates.

```

610     X = [ X1+(0:b/2-1)*width ...
611             (X1+X2)/2-(X2-X1-b*width)/2*cos(linspace(0,pi,nPatch-b)) ...
612             X2+(1-b/2:0)*width ];
613     else% nPatch==1
614         warning('nPatch==1 so placing one patch at the mean of Xlim')

```

<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??

```

615      X=mean(Xlim);
616    end

```

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

```

625  case 'usergiven'
626    X = reshape(Dom.X,1,[]);
627  end%switch Dom.type

```

Fourth, construct the microscale grid in each patch, centred about the given mid-points  $X$ . Reshape the grid to be 4D to suit dimensions (micro,Vars,Ens,macro).

```

637  xs = dx*( (1:nSubP)-mean(1:nSubP) );
638  patches.x = reshape( xs'+X ,nSubP,1,1,nPatch);

```

### 3.1.4 Set ensemble inter-patch communication

For EdgyInt or centre interpolation respectively,

- the right-edge/centre realisations  $1:n_{\text{Ensem}}$  are to interpolate to left-edge  $\text{le}$ , and
- the left-edge/centre realisations  $1:n_{\text{Ensem}}$  are to interpolate to  $\text{re}$ .

$\text{re}$  and  $\text{li}$  are ‘transposes’ of each other as  $\text{re}(\text{li})=\text{le}(\text{ri})$  are both  $1:n_{\text{Ensem}}$ . 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()`.

```

668 nE = patches.nEnsem;
669 patches.le = 1:nE;
670 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:  $n_{\text{SubP}}$  times should be enough. If  $\text{cs}$  is more then 2D, then the higher-dimensions are reshaped into the 2nd dimension.

```

682 if ~isempty(cs)
683   [mx,nc] = size(cs);
684   nx = nSubP(1);
685   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.

---

```
693     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.

```
702     patches.nEnsem = mx;
703     patches.cs = nan(nx-1,nc,mx);
704     for i = 1:mx
705         is = (i:i+nx-2);
706         patches.cs(:,:,i) = cs(is,:);
707     end
708     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.

```
718     patches.le = mod((0:mx-1)' + mod(nx-2,mx),mx)+1;
719     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??

```
727 if ratio*patches.nEnsem>0.9, warning( ...
728 'Probably poor scale separation in ensemble of coupled phase-shifts')
729 scaleSeparationParameter = ratio*patches.nEnsem
730 end
```

End the two if-statements.

```
736 end%if-else nEnsem>1
737 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>

```
756 if patches.parallel
757 % theparpool=gcp()
758 spmd
```

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

```
765 pari = 1;
766 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

---

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

invoked inside an `spmd`-group in order for this to work—so we do not need `parallel` in argument list.

```
776     switch pari
777         case 1, patches.x=codistributed(patches.x,patches.codist);
778     otherwise
779         error('should never have bad index for parallel distribution')
780     end%switch
781 end%spmd
```

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

```
789 else% not parallel
790     if isfield(patches,'codist'), rmfield(patches,'codist'); end
791 end%if-parallel
```

**Fin**

```
800 end% function
```

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

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

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

### Input

- `u` is a vector/array of length `nSubP · nVars · nEnsem · nPatch` where there are `nVars · nEnsem` field values at each of the points in the  $n_{SubP} \times n_{Patch}$  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 `nSubP × nVars × nEnsem × nPatch`. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - `.x` is  $n_{SubP} \times 1 \times 1 \times n_{Patch}$  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 `nSubP · nVars · nEnsem · nPatch` 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()`.

```
82 sizeu = size(u);
83 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.

```
93 dudt=patches.fun(t,u,patches,varargin{:});
94 n=patches.nEdge;
```

```
95 dudt([1:n end-n+1:end],:,:,:,:) = 0;  
96 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 `nSubP · nVars · nEnsem · nPatch` where there are `nVars · nEnsem` field values at each of the points in the  $n_{\text{SubP}} \times n_{\text{Patch}}$  multiscale spatial grid.
- `patches` a struct largely set by configPatches1(), and which includes the following.
  - `.x` is  $n_{\text{SubP}} \times 1 \times 1 \times n_{\text{Patch}}$  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).
  - `.nEdge`, for each patch, the number of edge values set by interpolation at the edge regions of each patch (default is one).

---

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

- `.nEnsem` the number of realisations in the ensemble.
- `.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.

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

```

114 [nx,~,~,Nx] = size(patches.x);
115 nEnsem = patches.nEnsem;
116 nVars = round(numel(u)/numel(patches.x)/nEnsem);
117 assert(numel(u) == nx*nVars*nEnsem*Nx ...
118 , 'patchEdgeInt1: input u has wrong size for parameters')
119 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.

**Implement multiple width edges by folding** Subsample  $x$  coordinates, noting it is only differences that count *and* the microgrid  $x$  spacing must be uniform.

```

133 x = patches.x;
134 if patches.nEdge>1
135     nEdge = patches.nEdge;
136     x = x(1:nEdge:nx,:,:,:);
137     nx = nx/nEdge;
138     u = reshape(u,nEdge,nx,nVars,nEnsem,Nx);
139     nVars = nVars*nEdge;
140     u = reshape(permute(u,[2 1 3:5]) ,nx,nVars,nEnsem,Nx);
141 end%if patches.nEdge
```

**Staggered grid** 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> <sup>8</sup>

```

157 if patches.stag % transform by doubling the number of fields
158 error('Doubt staggered works following changes made in Oct 2023??')
159 v = nan(size(u)); % currently to restore the shape of u
```

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

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

<sup>8</sup> **ToDo:** Have not yet implemented multiple edge values for a staggered grid as I am uncertain whether it makes any sense.

```

160      u = [u(:,:, :,1:2:Nx) u(:,:, :,2:2:Nx)];  

161      stagShift = 0.5*[ones(1,nVars) -ones(1,nVars)];  

162      iV = [nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field  

163      patches.ratio = patches.ratio/2; % ratio effectively halved  

164      Nx = Nx/2; % halve the number of patches  

165      nVars = nVars*2; % double the number of fields  

166      else % the values for standard spectral  

167          stagShift = 0;  

168          iV = 1:nVars;  

169      end%if patches.stag

```

### 3.3.1 Interpolate in the x-direction

Only use the interior values of the fields for interpolating to the edges.

```
179 u = u(2:nx-1,:,:,:);
```

Interpolate in turn, the edge or mid-patch edges normal to the  $x, y$ -directions, in this way we naturally fill-in corner values.

```

187 u = patchEdgeIntCore(1,u,x,patches,stagShift ...  

188     ,1,nx,nVars,nEnsem,1,Nx,1,patches.le,patches.ri);

```

Restore array  $u$  to its original shape.

```
194 u = reshape(u,nx,nVars,nEnsem,Nx);
```

### Unfold staggered Grid

```

202 if patches.stag % transform by doubling the number of fields  

203     patches.ratio = patches.ratio*2; % restore ratio  

204 end

```

**Unfold multiple edges** No need to restore  $x$ .

```

213 if patches.nEdge>1  

214     nVars = nVars/nEdge;  

215     u = reshape( u ,nx,nEdge,nVars,nEnsem,Nx);  

216     nx = nx*nEdge;  

217     u = reshape( permute(u,[2 1 3:5]) ,nx,nVars,nEnsem,Nx);  

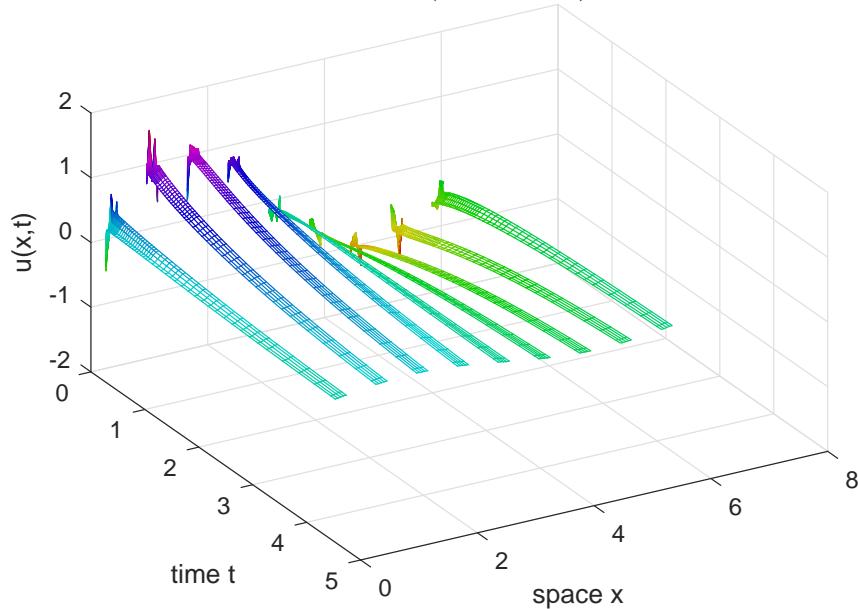
218 end%if patches.nEdge

```

Fin, returning the 4D array of field values.

```
225 end% function patchEdgeInt1
```

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

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

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

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

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

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

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 ifUrCf2eps([mfilename 'CtsU'])
106 title('space-time solution $u(x,t)$ on patches in space')
```

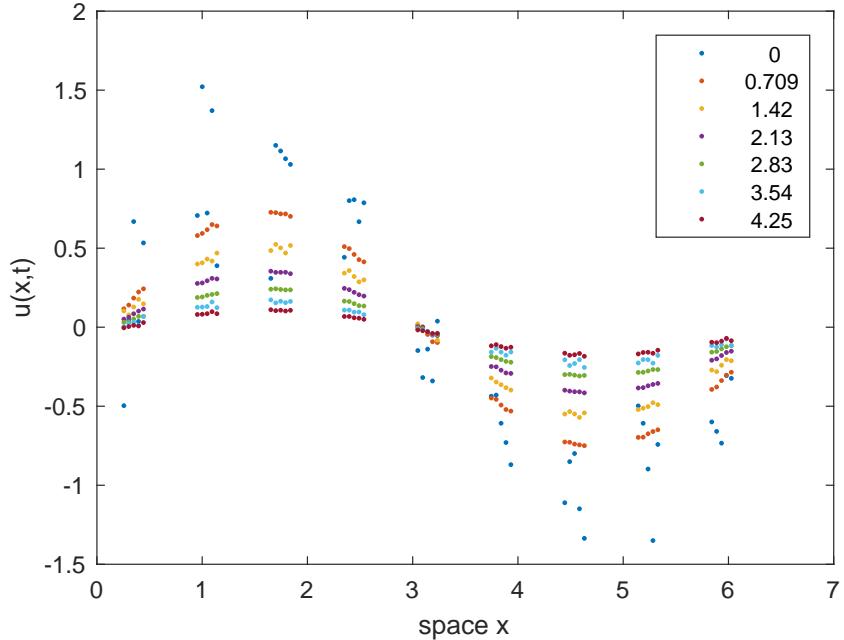
The code may invoke this integration interface.

```

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');
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

*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.*



(Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.4.3), as illustrated by Figure 3.4.

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

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

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

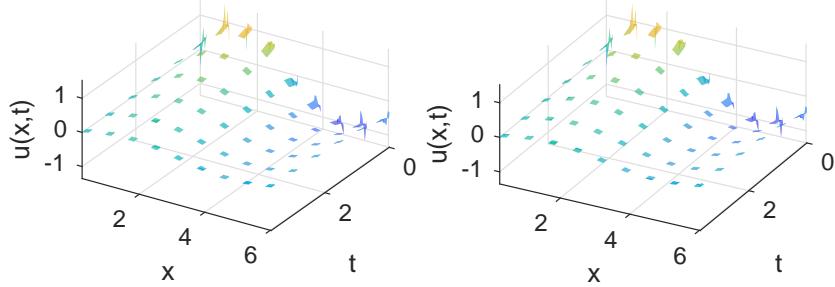
```
142 u0([1 end], :) = nan;
```

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.

```
154 ts = linspace(0, 2/cHomo, 7)
155 bT = 3*( ratio*Len/nPatch )^2/cHomo
156 %addpath('..../ProjInt')
157 [us, tss, uss] = PIRK2(@heteroBurst, ts, u0(:), bT);
```

Plot the macroscale predictions to draw Figure 3.4.

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.



```

164 figure(2),clf
165 plot(xs(:,us,'.')
166 ylabel('$u(x,t)$'), xlabel('space $x$')
167 legend(num2str(ts',3))
168 ifOurCf2eps([mfilename 'U'])
169 title('spatial structure of $u(x,t)$ in patches at selected times')

```

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

```

184 figure(3),clf
185 for k = 1:2, subplot(2,2,k)
186 surf(tss,xs(:,uss', 'EdgeColor','none')
187 ylabel('$x$'), xlabel('$t$'), zlabel('$u(x,t)$')
188 axis tight, view(126-4*k,45)
189 title('stereo of $u(x,t)$ on space-time patches')
190 end
191 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
```

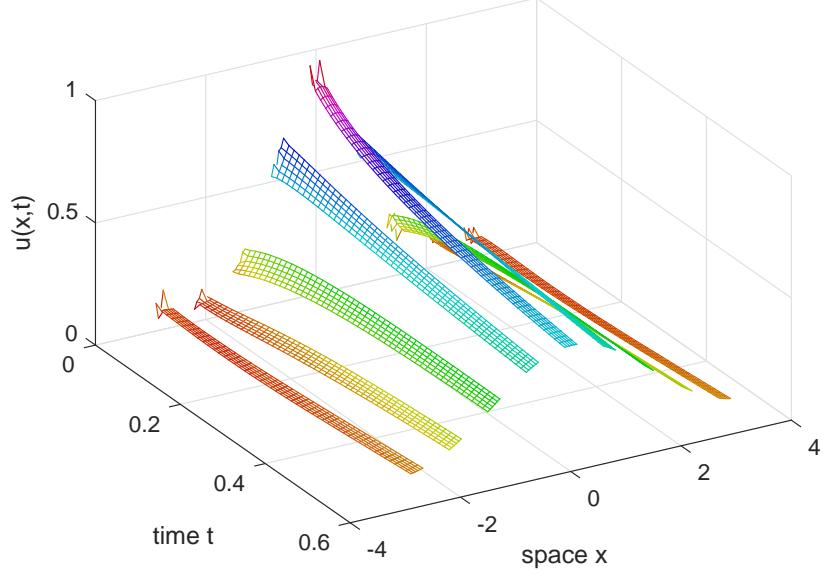
### 3.4.3 heteroBurst(): a burst of heterogeneous diffusion

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

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

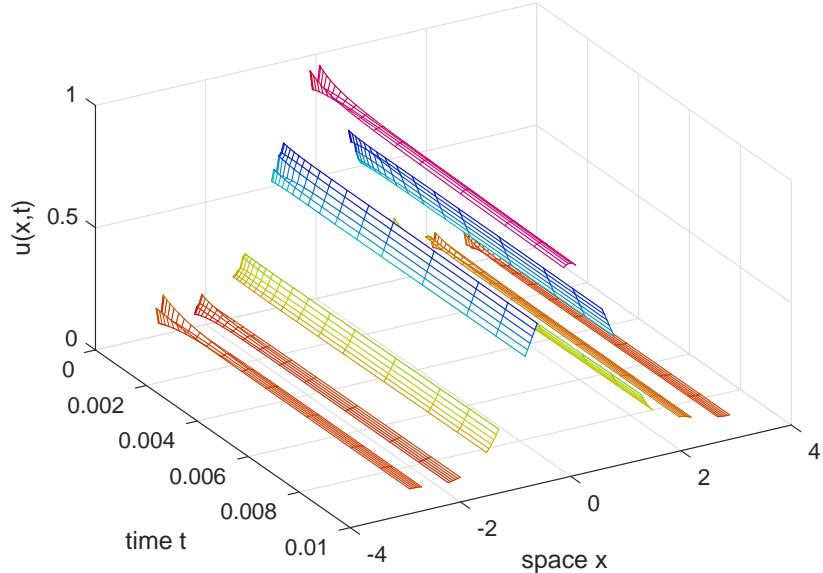
in terms of the centred difference operator  $\delta$ . The system has a microscale heterogeneity via the coefficients  $c_{i+1/2}$  which we assume to have some given known periodicity. Figure 3.6 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

#### 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  $\mapsto$  patchSys1  $\mapsto$  heteroDiff
3. plot the simulation

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



#### 4. use patchSys1 to explore the Jacobian

First establish the microscale heterogeneity has micro-period `mPeriod` on the lattice, and random log-normal values, albeit normalised to have harmonic mean one. This normalisation then means that macroscale diffusion on a domain of length  $2\pi$  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 switch p
196 case 1, title('short time of simulation showing transients')
197 case 2, title('simulation of $u(x,t)$ on patches in space')
198 end
199 end
200 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.

```

213 nPatch = 13
214 ratio = 0.01;
215
216 leadingEvals=[];
217 for ord=0:2:8
218     ordInterp=ord
219     configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
220         ,ord,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
221         ,'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.

```

231 u0 = zeros(nSubP,1,nEnsem,nPatch);
232 u0([1 end],:,:, :)=nan; u0=u0(:);
233 i=find(~isnan(u0));
234 nJ=length(i);
235 Jac=nan(nJ);
236 for j=1:nJ
237     u0(i)=((1:nJ)==j);
238     dudt=patchSys1(0,u0);
239     Jac(:,j)=dudt(i);
240 end
241 nonSymmetric=norm(Jac-Jac')
242 assert(nonSymmetric<5e-9,'failed symmetry')
243 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

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
```

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.

```
284 [evecs,evals]=eig(Jac);
285 eval=-sort(-diag(real(evals)));
286 nZeroEv=sum(eval(:)>-1e-5)
287 leadingEvals=[leadingEvals eval(1:nPatch)];
294 end
295 disp(' spectral quadratic quartic sixth-order ...')
296 leadingEvals=leadingEvals
```

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

End of the main script.

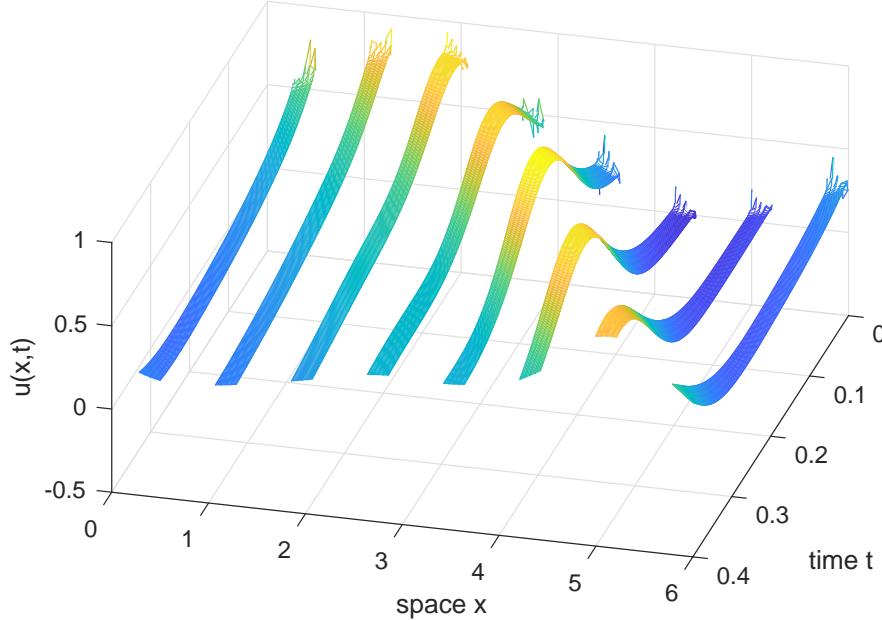
### 3.5.2 `heteroDiff()`: heterogeneous diffusion

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

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  $\mapsto$  patchSys1  $\mapsto$  burgersMap
3. process results

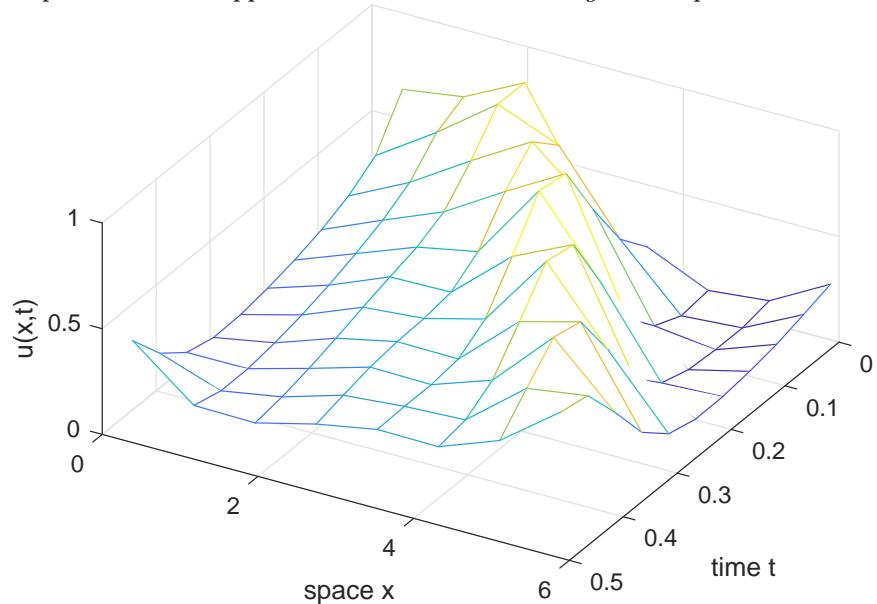
Establish global data struct for the microscale Burgers' map (Section 3.6.3) solved on  $2\pi$ -periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth-order interpolation provides edge-values that couple the patches.

```

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);

```

*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.*



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 ifOrCf2eps([mfilename 'MapU'])
```

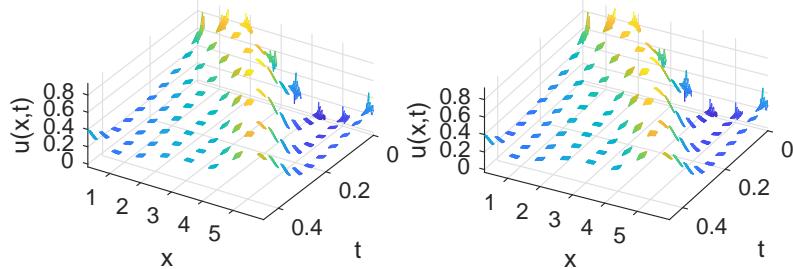
### 3.6.2 Alternatively use projective integration

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  $\mapsto$  burgerBurst  $\mapsto$  patchSys1  $\mapsto$  burgersMap
3. process results

*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.*



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
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 ifOurCf2eps([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 ifOurCf2eps([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.

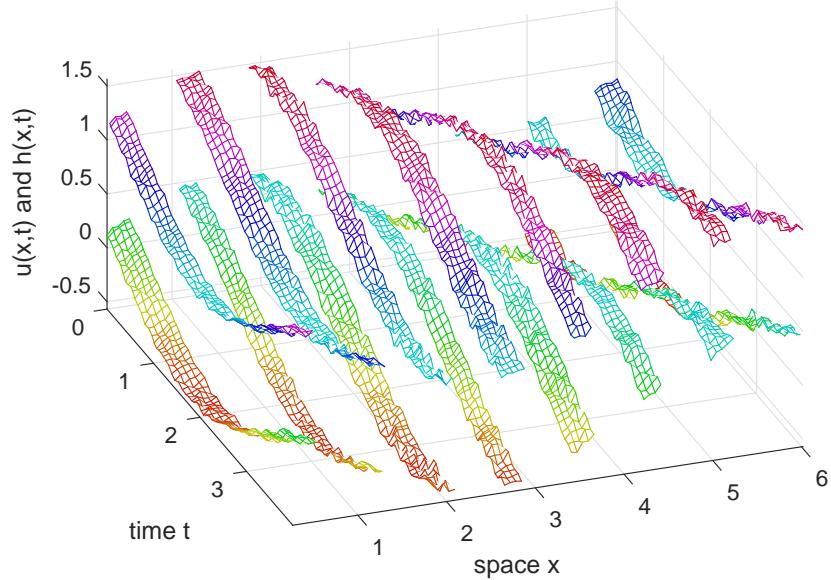
```

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

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.

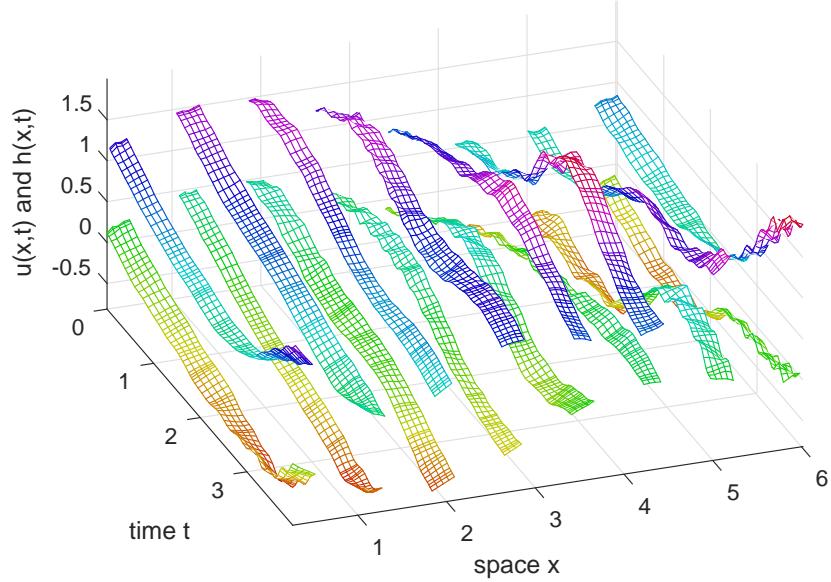
This approach, based upon the differential equations coded in Section 3.7.2, may be adapted by a user to a wide variety of 1D wave and wave-like systems. For example, the differential equations of Section 3.7.3 that describe the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived 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

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

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

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.



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  $\mapsto$  patchSys1  $\mapsto$  idealWavePDE
3. process results
4. ode15s  $\mapsto$  patchSys1  $\mapsto$  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] = ode0cts(@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  $\mathbf{Ut}$  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  $\dot{h}_{ij}$ .

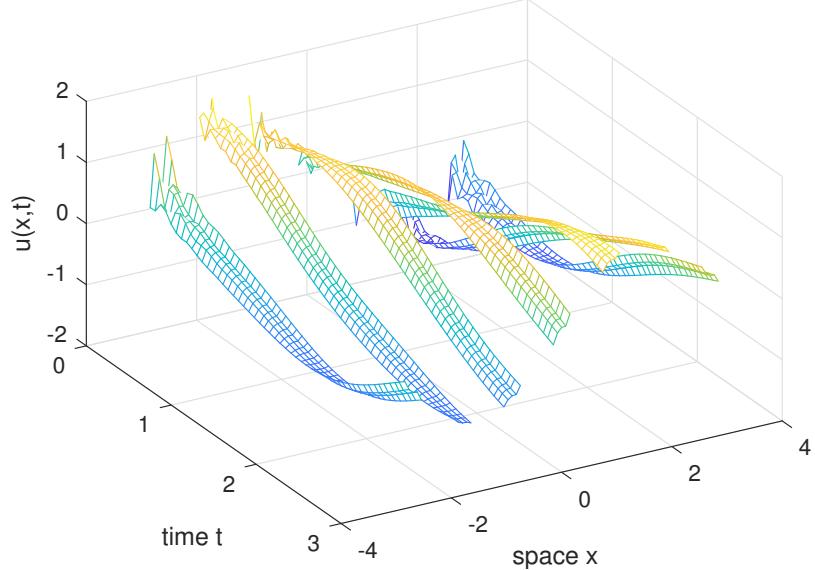
```

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

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

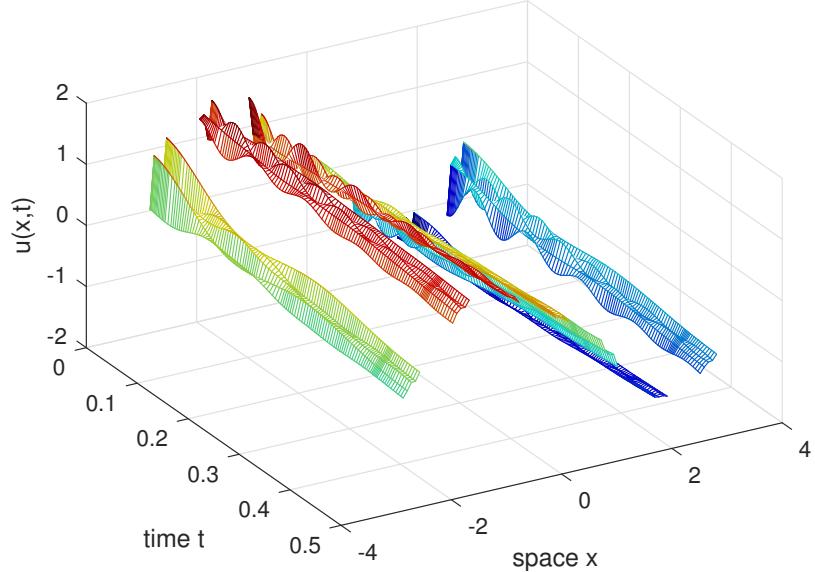
in terms of the centred difference operator  $\delta$ . The system has a microscale heterogeneity via the coefficients  $c_{i+1/2}$  which we assume to have some given known periodicity. [Figure 3.13](#) shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

#### 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`  $\mapsto$  `patchSys1`  $\mapsto$  `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  $\mathbf{ut}$ :

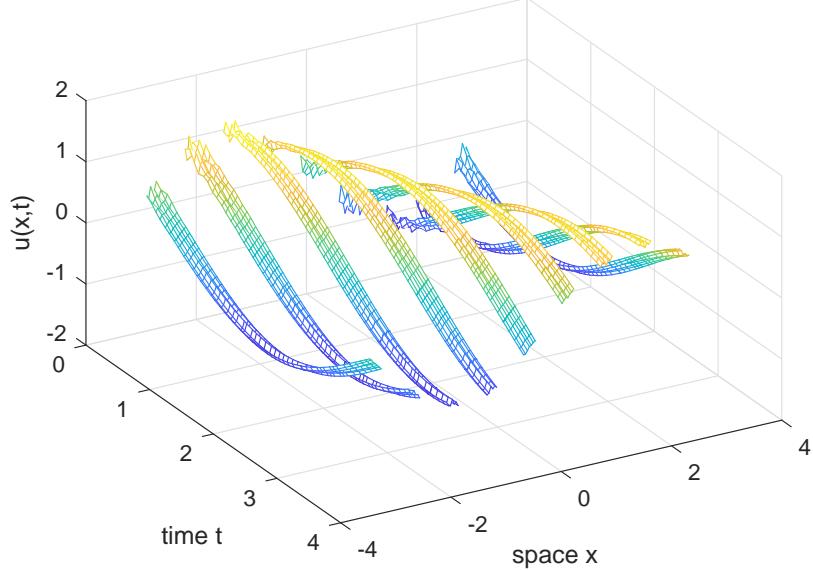
$$\frac{\partial u_{ij}}{\partial t} = v_{ij}, \quad \frac{\partial v_{ij}}{\partial t} = \frac{1}{dx^2} \delta[c_{i-1/2} \delta u_{ij}] + \frac{0.02}{dx^2} \delta^2 v_{ij}.$$

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

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,  $\nu \delta^2 u_i / d^2$ , 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  $\mapsto$  patchSys1  $\mapsto$  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 ...
127     , 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 ...
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);

```

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
```

```
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  $ut$ .

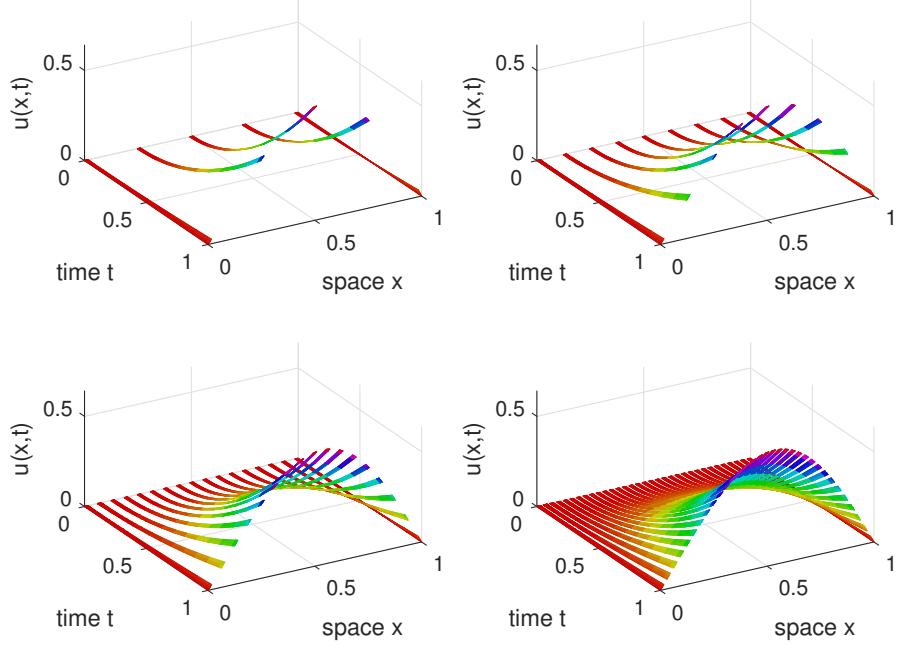
```

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

```

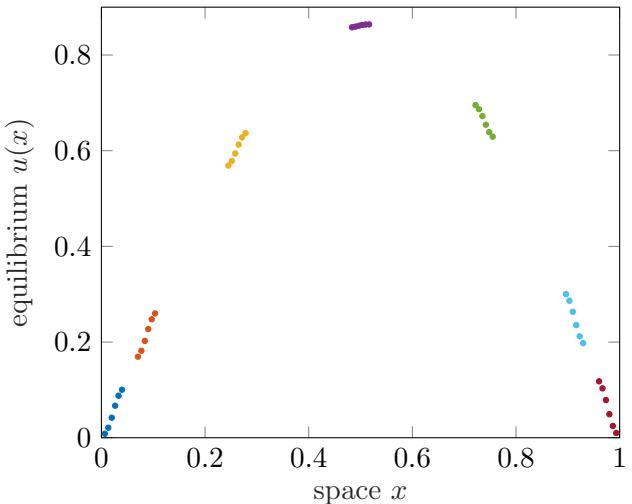
Code the microscale forced diffusion.

```

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));
```

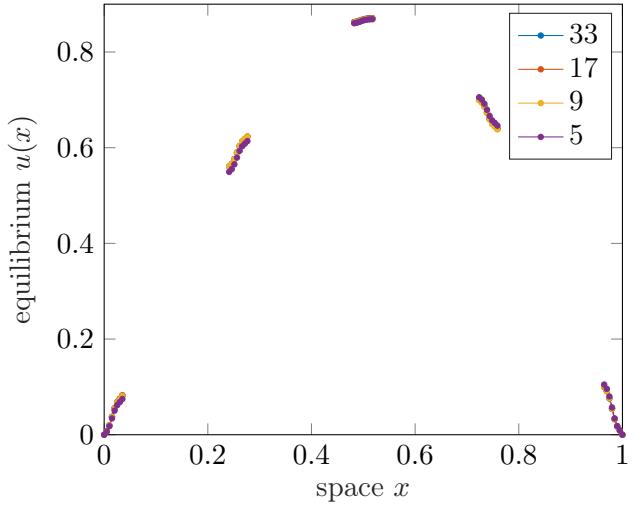
Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` ([Section 3.19](#)).

```
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:

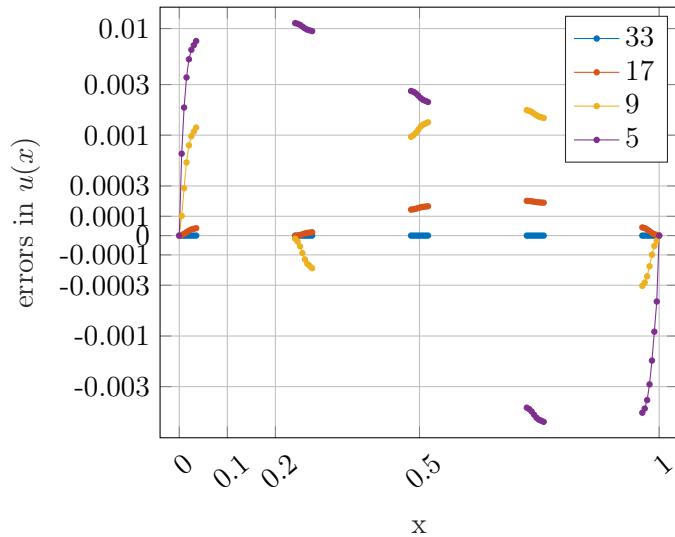
	$N =$	5	9	17	33	65
equispace	second-order	8E-3	1E-2	1E-2	4E-3	9E-4
equispace	fourth-order	2E-3	7E-4	1E-4	9E-6	5E-7
equispace	sixth-order	2E-3	2E-5	4E-7	1E-8	2E-10
chebyshev	second-order	4E-2	6E-2	3E-2	2E-2	2E-2
chebyshev	fourth-order	9E-4	3E-3	6E-4	3E-4	2E-4
chebyshev	sixth-order	9E-4	3E-5	1E-5	4E-6	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>9</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>9</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  $\epsilon$ , 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 3.19](#)).

```

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

```

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

```

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

```

Concatenate the solution on common patches into stores.

```

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

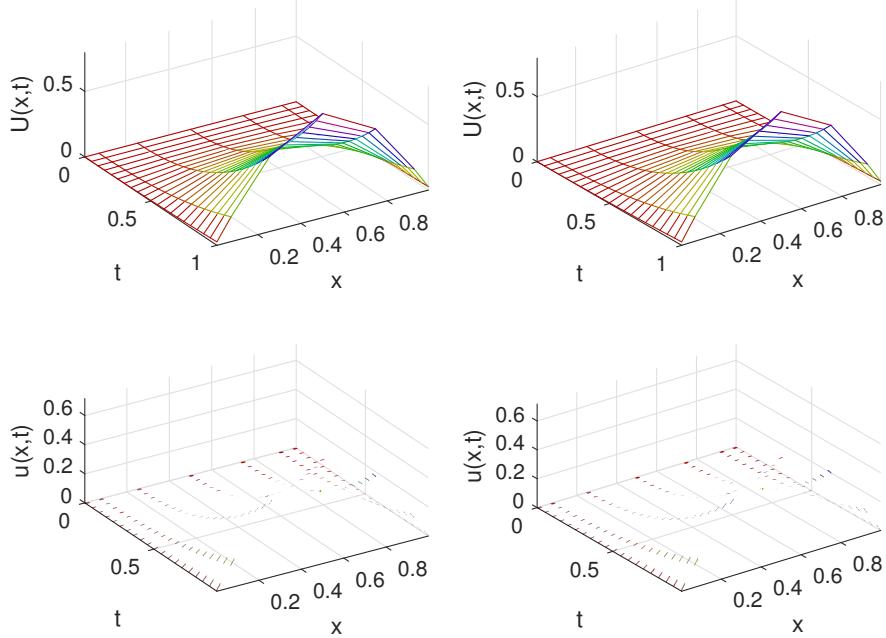
    Reshape solution field.

322 figure(1),clf
323 plot(x(:,u,'.-'), legend(num2str(nPs))
324 xlabel('space $x$'), ylabel('equilibrium $u(x)$')
325 if0urCf2tex([mfilename 'us'])%optionally save
```

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

```
333 err = u(:,1)-u;
334 figure(2), clf
335 plot(x(:,err,'.-'); legend(num2str(nPs))
336 quasiLogAxes(10,sqrt(prod(meanAbsErrs(2:3))))
337 xlabel('space $x$'), ylabel('errors in $u(x)$')
338 if0urCf2tex(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('space $x$'), xlabel('time $t$'), 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('space $x$'), xlabel('time $t$'), zlabel('$u(x,t)$')
236 colormap(0.7*hsv), axis tight, view(62-4*k,45)
237 end
238 if0urCf2eps(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>10</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

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

in terms of the centred difference operator  $\delta$ . 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>11</sup>

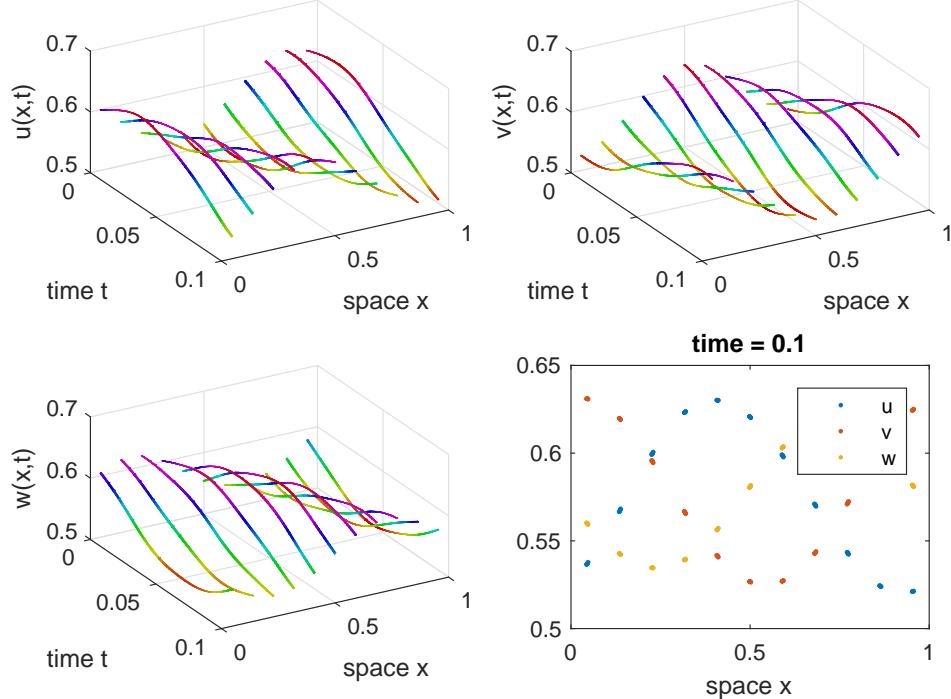
```
89 global alpha ex5p1
90 ex5p1 = 0; % set to 1 for L&O example of p.27
91 alpha = 0.001 % phenomenological damping parameter
```

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

<sup>10</sup> Recall  $a \times (b \times c) = (a \cdot c)b - (a \cdot b)c$

<sup>11</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  $\mapsto$  patchSys1  $\mapsto$  heteroLanLif1D
3. plot the simulation

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

```
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')
```

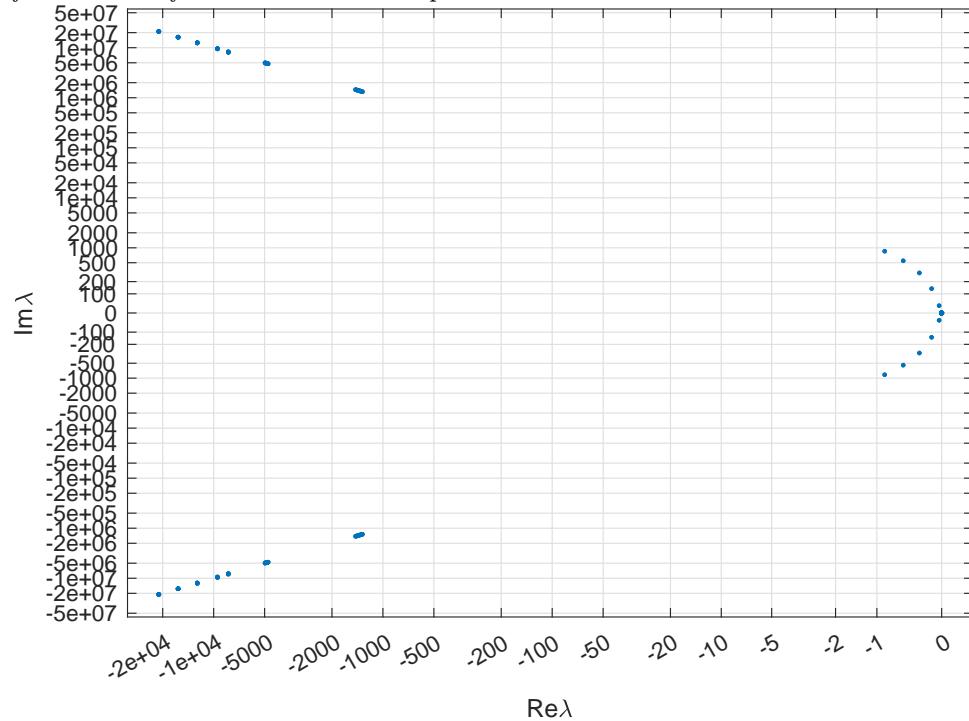
**Plot space-time surface of the simulation** Choose whether to save some plots, or not.

```

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 plot(real(eval),imag(eval),'.');
315 xlabel('$\Re\lambda$'), ylabel('$\Im\lambda$')
316 quasiLogAxes(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)):

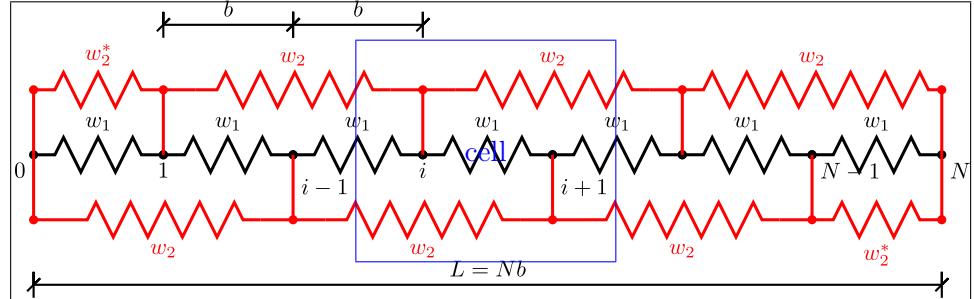
```

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  $i$ ,

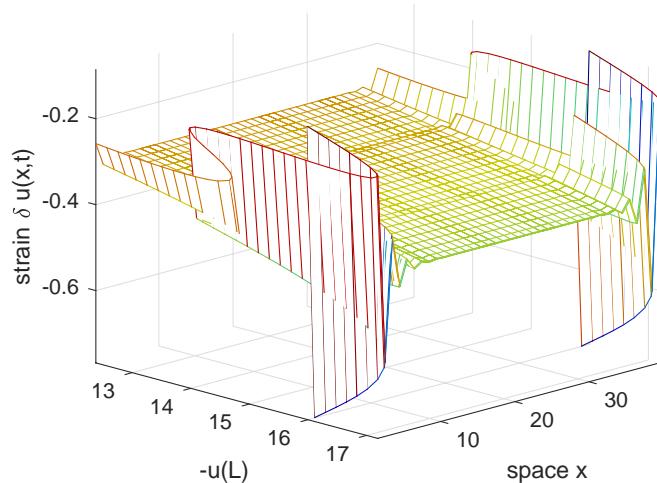
$$\begin{aligned} \epsilon_i^p &:= \frac{1}{pb}(u_{i+p/2} - u_{i-p/2}), & \sigma_i^p &:= w_p'(\epsilon_i^p), \\ \frac{\partial^2 u_i}{\partial t^2} &= \sum_{p=1}^2 \frac{1}{pb}(\sigma_{i+p/2}^p - \sigma_{i-p/2}^p), & w_p'(\epsilon) &:= \epsilon - M_p \epsilon^3 + \epsilon^5. \end{aligned} \quad (3.9)$$

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

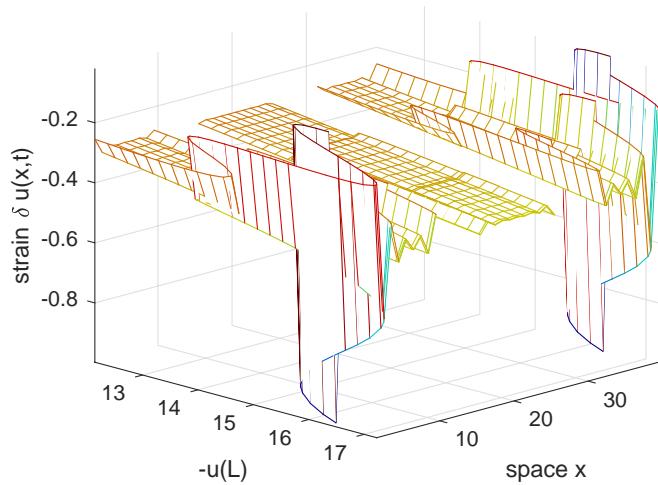
points and neutral saddle equilibria in this same regime (see [Figure 3.27\(b\)](#)). I have not yet followed any of the branches.

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.

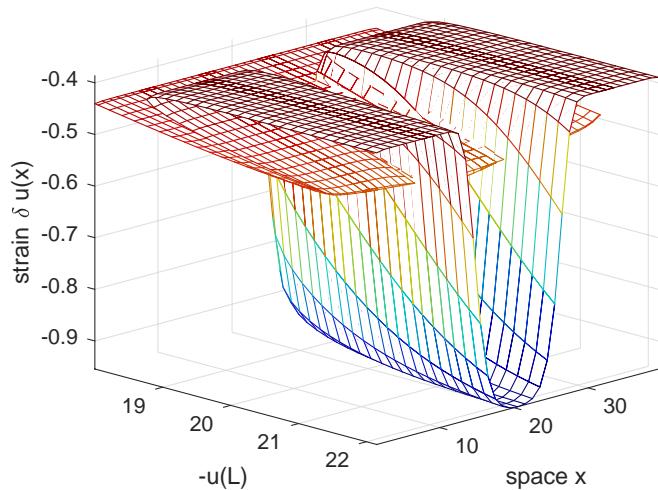
Importantly, MatCont is about *ten times quicker to execute on the patches* than on the full domain code. This speed-up indicates that on larger scale problems the patch scheme could be very useful in continuation explorations.

**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 '\n'])
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.

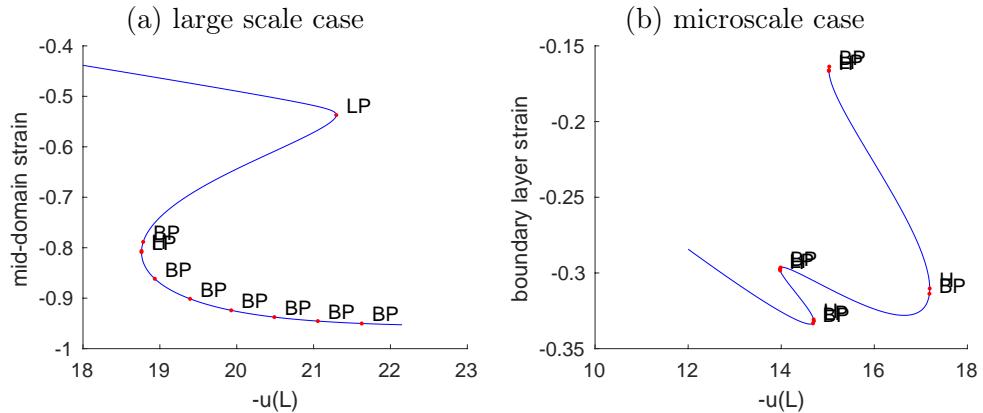
```
454 figure(5), clf()
455 mesh(muLP, xs(1:end-1),diff(us))
456 view(45,20), colormap(0.8*jet), axis tight
457 xlabel('$-u(L)$'), ylabel('space $x$'), zlabel('strain $\delta u(x)$')
458 ifOurCf2eps(['Comb22diffu' num2str(theCase)], [12 9])%optionally save
```

**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
```

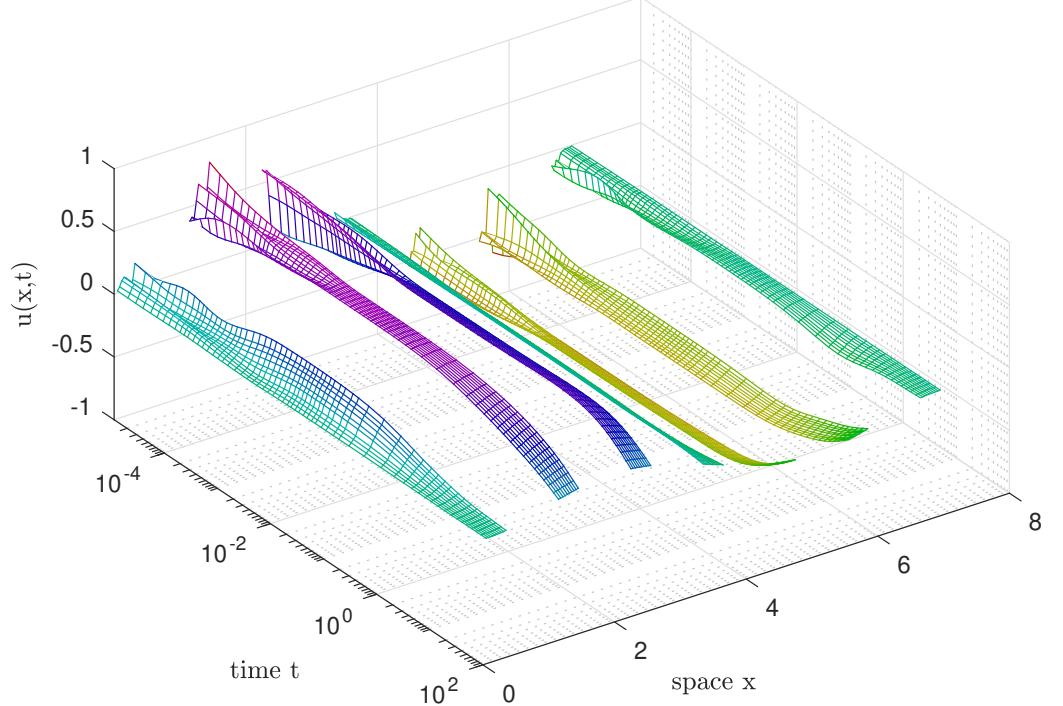
Set corresponding nonlinear stresses

```
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
```

*Figure 3.28:* hyper-diffusing field  $u(x, t)$  in the patch scheme applied to microscale heterogeneous hyper-diffusion (Section 3.16). The log-time axis shows:  $t < 10^{-2}$ , rapid decay of sub-patch micro-structure;  $10^{-2} < t < 1$ , meso-time quasi-equilibrium; and  $1 < t < 10^2$ , slow decay of macroscale structures.



### 3.16 hyperDiffHetero: simulate a heterogeneous hyper-diffusion PDE in 1D on patches

Figure 3.28 shows an example simulation in time generated by the patch scheme applied to a heterogeneous version of the hyper-diffusion PDE. That such simulations makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

We aim to simulate the heterogeneous hyper-diffusion PDE

$$u_t = -D[c_1(x)Du] \quad \text{where operator } D := \partial_x(c_2(x)\partial_x), \quad (3.10)$$

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

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

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

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

```

57 clear all
58 basename = mfilename
59 %global OurCf2eps, OurCf2eps=true %optional to save plots
60 nGap = 3 % controls size of gap between patches
61 nPtsPeriod = 5
62 dx = 0.5/nGap/nPtsPeriod

Create some random heterogeneous coefficients, log-uniform.

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

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

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

```

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

```

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

```

Plot the simulation in [Figure 3.28](#), using log-axis for time so we can see a little of both micro- and macro-dynamics.

```

116 figure(1),clf
117 xs([1:2 end-1:end],:) = nan;
118 t0=min(find(ts>1e-5));
119 mesh(ts(t0:3:end),xs(:,),us(t0:3:end,:)), view(55,50)
120 colormap(0.7*hsv)
121 xlabel('time $t$'), ylabel('space $x$'), zlabel('$u(x,t)$')
122 ca=gca; ca.XScale='log'; ca.XLim=ts([t0 end]);
123 ifOurCf2eps([basename 'Uxt'])

```

Fin.

### 3.16.1 Heterogeneous hyper-diffusion PDE inside patches

As a microscale discretisation of hyper-diffusion PDE (3.10)  $u_t = -D[c_1(x)Du]$ , where heterogeneous operator  $D = \partial_x(c_2(x)\partial_x)$ .

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

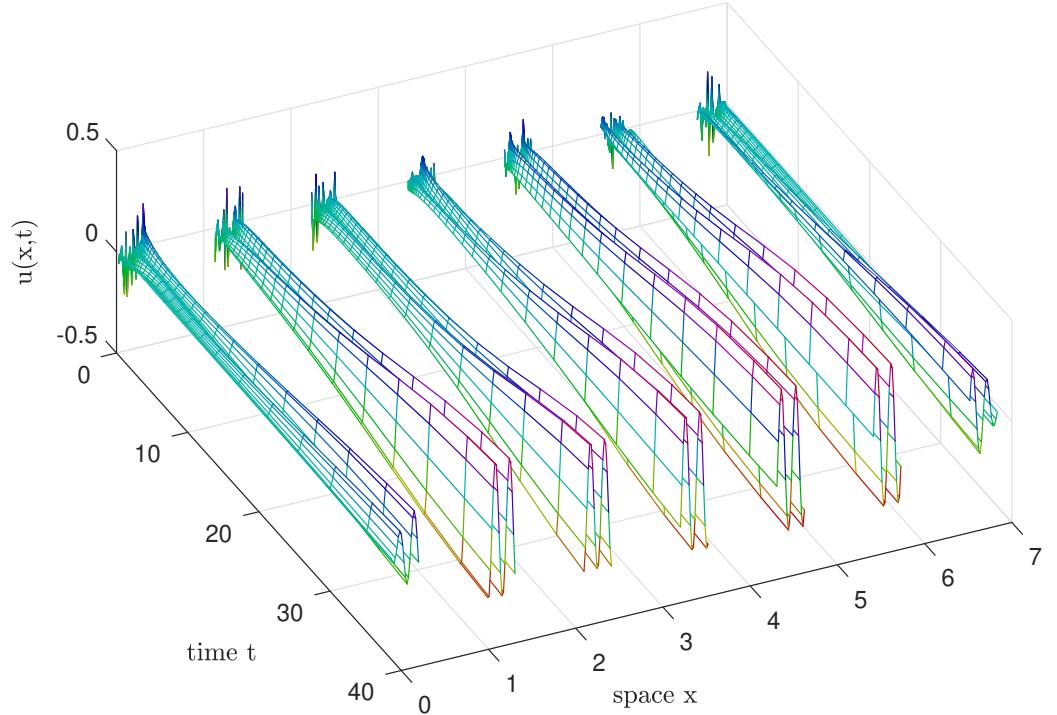
Code Dirichlet boundary conditions of zero function and derivative at left-end of left-patch, and right-end of right-patch. For slightly simpler coding, squeeze out the two singleton dimensions.

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

Here code straightforward centred discretisation in space.

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

*Figure 3.29: the pattern forming field  $u(x, t)$  in the patch (gap-tooth) scheme applied to a microscale discretisation of the Swift–Hohenberg PDE (Section 3.17). Physically we see the rapid decay of much microstructure, but also the meso-time growth of sub-patch-scale patterns, wavenumber  $k_0$ , that are modulated over the inter-patch distances and over long times.*



### 3.17 SwiftHohenbergPattern: patterns of the Swift–Hohenberg PDE in 1D on patches

Figure 3.29 shows an example simulation in time generated by the patch scheme applied to the patterns arising from the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by [Bunder et al. \(2017\)](#) who proved that the scheme is accurate when the number of points in a patch is just more than a multiple of the periodicity of the pattern.

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

$$\partial_t u = -(1 + \partial_x^2/k_0^2)^2 u + Ra u - u^3, \quad (3.11)$$

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

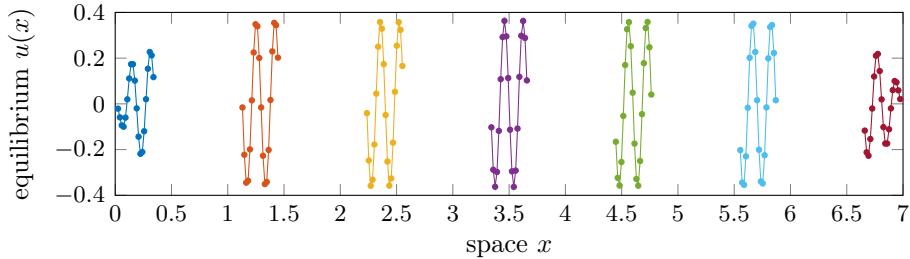
Set the desired microscale periodicity of the emergent pattern.

```

52 clear all, close all
53 %global OurCf2eps, OurCf2eps=true %optional to save plots

```

*Figure 3.30: an equilibrium of the Swift–Hohenberg PDE on seven patches in 1D space. In the sub-patch patterns, there is a small phase shift in the patterns from patch to patch. And the amplitude of the pattern has to go to ‘zero’ at the boundaries.*



```

54 Ra = 0.1 % Ra>0 leads to patterns
55 nGap = 3
56 %waveLength = 0.496688741721854 /nGap %for nPatch==5
57 waveLength = 0.497630331753555 /nGap %for nPatch==7
58 %waveLength = 0.5 /nGap %for periodic case
59 nPtsPeriod = 10
60 dx = waveLength/nPtsPeriod
61 k0 = 2*pi/waveLength

```

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

```

72 nPatch = 7
73 nSubP = 2*nPtsPeriod+4
74 %nSubP = 2*nGap*nPtsPeriod+4 % full-domain
75 Len = nPatch;
76 ordCC = 4;
77 dom.type='equispace';
78 dom.bcOffset=0.5
79 patches = configPatches1(@SwiftHohenbergPDE,[0 Len],dom ...
80 ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
81 xs=squeeze(patches.x);

```

### 3.17.0.1 Find equilibrium with `fsolve`

Start the search from some guess.

```

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

```

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

```

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

```

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` ([Section 3.19](#)).

```

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

```

**Plot the equilibrium** see [Figure 3.30](#).

```

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

```

### 3.17.0.2 Simulate in time

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

```

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

```

Plot the simulation in [Figure 3.29](#).

```

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

```

Fin.

### 3.17.1 The Swift–Hohenberg PDE and BCs inside patches

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

```

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

```

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

```

198     u = squeeze(u);
199     u(1:2,1)=0;
200     u(end-1:end,end)=0;

```

Here code straightforward centred discretisation in space.

```

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

```

### 3.17.2 theRes(): wrapper function to zero for equilibria

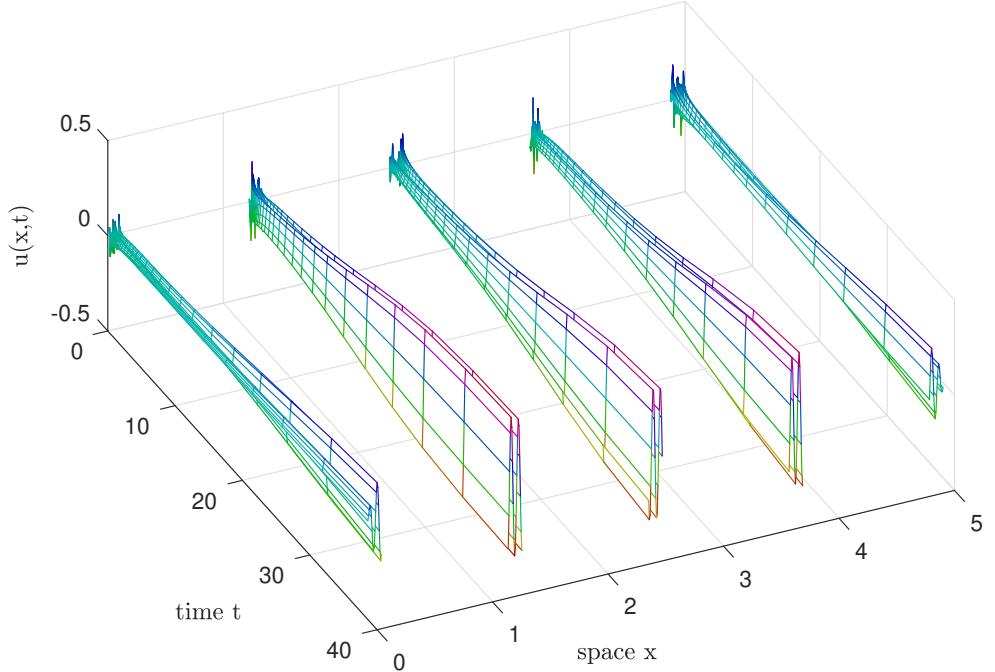
This functions converts a vector of values into the interior values of the patches, then evaluates the time derivative of the system at time zero, and returns the vector of patch-interior time derivatives.

```

225 function f=theRes(u,patches,k0,Ra)
226 v=nan(size(patches.x));
227 v(patches.i) = u;
228 f = patchSys1(0,v(:,),patches,k0,Ra);
229 f = f(patches.i);
230 end%function theRes

```

*Figure 3.31: the field  $u(x, t)$  in the patch (gap-tooth) scheme applied to microscale heterogeneous Swift–Hohenberg PDE (Section 3.18). The heterogeneous coefficients are approximately uniform over  $[0.9, 1.1]$ . This heterogeneity has no noticeable affect on the simulation.*



### 3.18 SwiftHohenbergHetero: patterns of a heterogeneous Swift–Hohenberg PDE in 1D on patches

Figure 3.31 shows an example simulation in time generated by the patch scheme applied to the patterns arising from a heterogeneous version of the Swift–Hohenberg PDE. That such simulations of patterns makes valid predictions was established by Bunder et al. (2017) who proved that the scheme is accurate when the number of points in a patch is tied to a multiple of the periodicity of the pattern.

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

$$\partial_t u = -D[c_1(x)Du] + Ra u - u^3, \quad D := 1 + \partial_x[c_2(x)\partial_x \cdot]/k_0^2, \quad (3.12)$$

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

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

Consider the spectrum in the symmetric cases of periodic patches (based upon only the cases  $N = 5, 7$ ). There are  $2N$  small eigenvalues, separated by a gap from the rest. In the homogeneous case, these occur as  $N$  pairs. With

small heterogeneity, they appear to split into  $N - 1$  pairs, and two distinct. With stronger heterogeneity (say 0.5), they *often* appear to also split into two clusters, each of  $N$  eigenvalues, with one small-valued cluster, and one meso-valued cluster—curious. Further analysis with sparse approximation of the invariant spaces suggests the following:

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

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

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

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

```

102 clear all
103 %global OurCf2eps, OurCf2eps=true %optional to save plots
104 basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
105 Ra = 0.1 % Ra>0 leads to patterns
106 nGap = 8 % controls size of gap between patches
107 waveLength = 0.496688741721854 /nGap %for nPatch==5
108 %waveLength = 0.497630331753555 /nGap %for nPatch==7
109 %waveLength = 0.5 /nGap %for periodic case
110 nPtsPeriod = 10
111 dx = waveLength/nPtsPeriod
112 k0 = 2*pi/waveLength

```

Create some random heterogeneous coefficients.

```

119 heteroVar = 0.99*[1 1] % must be <2
120 cl = 1./(1-heteroVar/2+heteroVar.*rand(nPtsPeriod,2));

```

```

121 cRange = quantile(c1,0:0.5:1)

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

133 nPatch = 5
134 nSubP = 2*nPtsPeriod+4 % +2 for not-edgyInt
135 %nSubP = 2*nGap*nPtsPeriod+4 % approx full-domain
136 Len = nPatch;
137 ordCC = 0;
138 dom.type='equispace';
139 dom.bcOffset=0.5
140 patches = configPatches1(@heteroSwiftHohenbergPDE,[0 Len],dom ...
141 ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2 ...
142 , 'hetCoeffs',c1);
143 xs=squeeze(patches.x);

```

### 3.18.0.1 Explore the Jacobian

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

```

155 fprintf('\n**** Explore the Jacobian\n')
156 u0 = 0*patches.x;
157 u0([1:2 end-1:end],:) = nan;
158 patches.i = find(~isnan(u0));
159 nVars = numel(patches.i)
160 Jac = nan(nVars);
161 for j=1:nVars
162     Jac(:,j)=theRes((1:nVars)==j,patches,k0,0,0);
163 end

```

Check on the symmetry of the Jacobian

```

169 nonSymmetric = norm(Jac-Jac')
170 Jac(abs(Jac)<1e-12)=0;
171 antiJac = Jac-Jac';
172 antiJac(abs(antiJac)<1e-12)=0;
173 figure(6),clf
174 spy(Jac,'.'),hold on, spy(antiJac,'rx'),hold off
175 if nonSymmetric>5e-9, warning('failed symmetry'),
176 else Jac = (Jac+Jac')/2; %tweak to symmetry
177 end

```

Compute eigenvalues and eigenvectors.

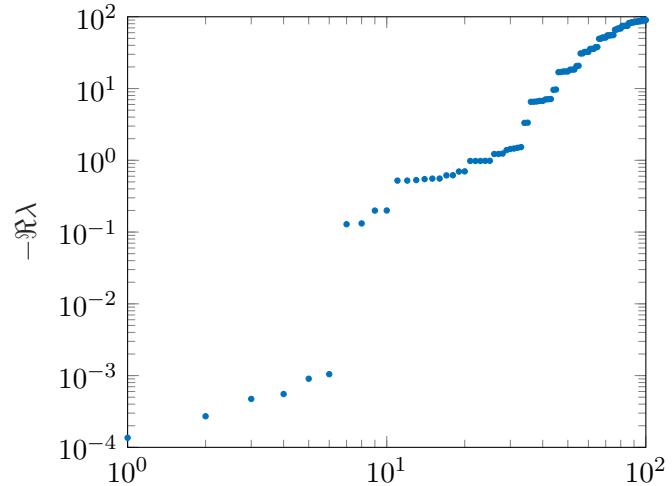
```

183 figure(5),clf
184 [evec,mEval] = eig(-Jac , 'vector');

```

Figure 3.32:

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



```

185 [~,j]=sort(real(mEval));
186 mEval=mEval(j); evec=evec(:,j);
187 loglog(real(mEval),'.')
188 ylabel('$-\text{Re}\lambda$')
189 ifOurCf2tex([basename 'Eval'])%optionally save

```

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

```

210 logGaps=diff(log10(real(mEval)));
211 [~,j]=sort(-logGaps);
212 %someLogGaps=[logGaps(j(1:5)) j(1:5)]
213 if logGaps(j(2))<0.4*logGaps(j(1)), nSlow=j(1)
214 else nSlow=min( sort(j(1:2)) , 3*nPatch)
215 end
216 log10Gap=logGaps(nSlow)
217 smallEvals=-mEval(1:nSlow(end)+2)

```

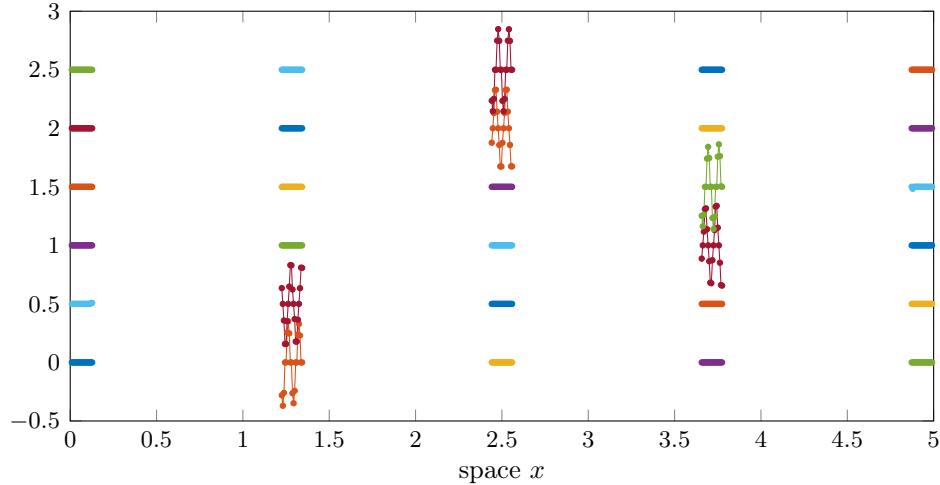
Second, make eigenvectors all real, sparsely approximate cluster modes via an algorithm developed from [Hu et al. \(2016\)](#), and plot. [Figure 3.33](#) shows that each pair of basis vectors are phase-shifted by  $90^\circ$ .

```

227 js=find(imag(mEval)>0);
228 evec(:,js)=imag(evec(:,js));
229 evec=real(evec);
230 if numel(nSlow)==1, S = spcart(evec(:,1:nSlow));
231 else S = spcart(evec(:,1:nSlow(1)));
232 S = [S spcart(evec(:,nSlow(1)+1:nSlow(2))) ];
233 end;
234 figure(3),clf
235 vStep=ceil(max(abs(S(:)))*10+1)/10
236 for j=1:nSlow(end)

```

*Figure 3.33: sparse approximations of the eigenvectors of the six slow modes of Figure 3.32. Plotted are sparse basis vectors for the invariant space spanned by the six slow eigenvectors: each basis vector shifted vertically to separate. Thus a fair approximation is that there are effectively two modes for each of the  $N - 2 = 3$  interior patches.*



```

237     u0(patches.i)=S(:,j);
238     plot(xs,vStep*(j-1)+squeeze(u0),'.-'),hold on
239   end
240   hold off, xlabel('space $x$')
241   ifOurCf2tex([basename 'Evec'])%optionally save

    Reorganise the eigenvectors to maybe clarify.

262   [i,j]=find(abs(S)>vStep/2);
263   j=find([1;diff(j)]);
264   [i,k]=sort(i(j));
265   figure(4)
266   for p=1:2
267     clf, subplot(2,1,1)
268     for j=p:2:numel(k)
269       u0(patches.i)=S(:,k(j));
270       plot(xs,squeeze(u0),'.-'),hold on
271     end% for j
272     hold off, xlabel('space $x$')
273     ifOurCf2tex([basename 'Evec' num2str(p)])%optionally save
274   end%for p

```

### 3.18.0.2 Find an equilibrium with fsolve

Start the search from some guess.

```

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

```

But set the pairs of patch-edge values to Nan in order to use `patches.i` to

Figure 3.34: sparse basis approximations for the invariant subspace of the six slow modes of Figure 3.32. A replot of Figure 3.33 but with three of the basis vectors superimposed in each of the two panels.

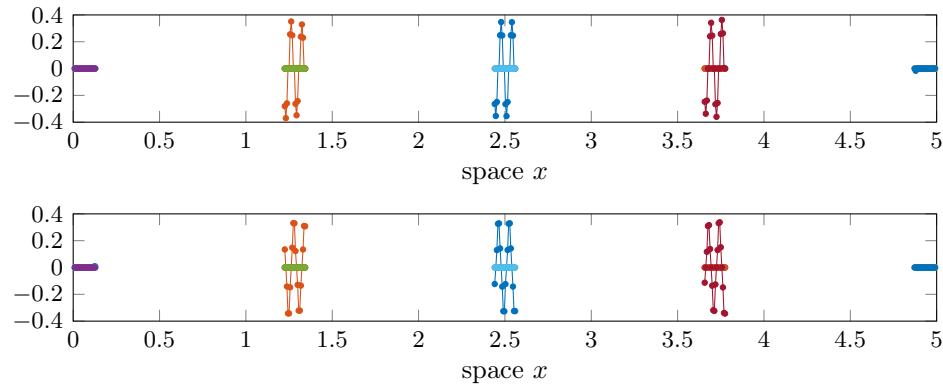
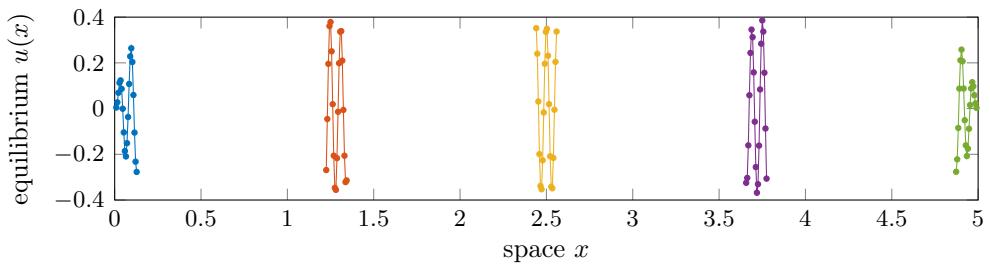


Figure 3.35: an equilibrium of the heterogeneous Swift–Hohenberg PDE determined by the patch scheme



index the interior sub-patch points as they are the variables.

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

Seek the equilibrium, and report the norm of the residual, via the generic patch system wrapper `theRes` (Section 3.18.2).

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

**Plot the equilibrium** see Figure 3.35.

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

### 3.18.0.3 Simulate in time

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

```

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

```

Plot the simulation in [Figure 3.31](#).

```

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

```

Fin.

### 3.18.1 Heterogeneous SwiftHohenberg PDE+BCs inside patches

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

```

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

```

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

```

399     u = squeeze(u);
400     if ~patches.periodic
401         switch 1
402             case 1 % these are u=u_x=0
403                 u(1:2,1)=0;
404                 u(end-1:end,end)=0;
405             case 2 % these are u=u_xx=0
406                 u(1:2,1) = [-u(3,1); 0];
407                 u(end-1:end,end) = [0; -u(end-2,end)];
408         end% case
409     end%if

```

Here code straightforward centred discretisation in space.

```

415     ut = nan+u;          % preallocate output array
416     v = u(2:end-1,:)+diff(patches.cs(:,2).*diff(u))/dx^2/k0^2;

```

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

### 3.18.2 theRes(): a wrapper function

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

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

### 3.19 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
```

### 3.20 quasiLogAxes(): transforms current axes of plot(s) to quasi-log

This function transforms the plots in the current axes. It rescales specified coordinates and labels the axes and its 2D or 3D plot(s). 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. This nonlinear rescaling is useful in other scenarios also.

```
22 function quasiLogAxes(xScale,yScale,zScale,cScale)
```

**Input** This function rescales the *current axes* (you may need to invoke `set()` to change to the axes you require). It rescales the existing plots (do not do further plots into the same axes).

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

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

**Example** If invoked with no arguments, then execute an example plot and its transformation.

```
61 if nargin==0
62   % generate some data
63   n=99;  fast=(rand(n,1)<0.8);
64   z = -rand(n,1).*(1+1e3*fast)+1i*randn(n,1).*(5+1e2*fast);
65   % plot data and transform axes
66   plot(real(z),imag(z),'.')
67   xlabel('real-part'), ylabel('imag-part')
68   title('un-transformed plot, pausing for 3 secs'), pause(3)
69   quasiLogAxes(1,10);
70   title('transformed plot')
71   return
72 end% example
```

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

```

81 if nargin<4, cScale=inf; end
82 if nargin<3, zScale=inf; end
83 if nargin<2, yScale=inf; end
84 if nargin<1, xScale=inf; end
85 assert(class(xScale)=="double" ...
86     , "May 2023 version of quasiLogAxes does not accept a handle")

```

Get current limits of the plot to use if the user has set them already. And also get the pointer to the axes and to the figure of the plot.

```

97 xlim0=xlim; ylim0=ylim; zlim0=zlim; clim0=caxis;
98 theAxes = gca; %get(handle(1),'parent');
99 theFig = get(theAxes,'parent');

```

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.

```

108 xFac=1e-99; yFac=xFac; zFac=xFac; cFac=xFac;
109 for kk=1:numel(theAxes.Children)
110     handle = theAxes.Children(kk);
111     for k=1:length(handle)
112         if ~isinf(xScale)
113             temp = asinh(handle(k).XData/xScale);
114             xFac = max(xFac, max(abs(temp(:)),[],'omitnan') );
115         end
116         if ~isinf(yScale)
117             temp = asinh(handle(k).YData/yScale);
118             yFac = max(yFac, max(abs(temp(:)),[],'omitnan') );
119         end
120         if ~isinf(zScale)
121             temp = asinh(handle(k).ZData/zScale);
122             zFac = max(zFac, max(abs(temp(:)),[],'omitnan') );
123         end
124         if ~isinf(cScale)
125             temp = asinh(handle(k).CData/cScale);
126             cFac = max(cFac, max(abs(temp(:)),[],'omitnan') );
127         end
128     end%for k
129 end%for kk
130 xFac=9/xFac; yFac=9/yFac; zFac=9/zFac; cFac=9/cFac;

```

Scale all the plot data in the axes. Give an error if it appears that the plot-data has already been transformed. Color data has to be transformed first because usually there is automatic flow from z-data to c-data.

```

140 xlim1=[Inf -Inf]; ylim1=xlim1; zlim1=xlim1; clim1=xlim1;
141 for kk=1:numel(theAxes.Children)
142     handle = theAxes.Children(kk);
143     for k=1:length(handle)
144         assert(~strcmp(handle(k).UserData,'quasiLogAxes'), ...
145             'Replot graph---it appears plot data is already transformed')

```

```

146      if ~isinf(cScale)
147          handle(k).CData = cFac*asinh(handle(k).CData/cScale);
148          climk=[min(handle(k).CData(:)) max(handle(k).CData(:))];
149          clim1=[min(climk(1),clim1(1)) max(climk(2),clim1(2))];
150      end
151      if ~isinf(xScale)
152          handle(k).XData = xFac*asinh(handle(k).XData/xScale);
153          xlimk=[min(handle(k).XData(:)) max(handle(k).XData(:))];
154          xlim1=[min(xlimk(1),xlim1(1)) max(xlimk(2),xlim1(2))];
155      end
156      if ~isinf(yScale)
157          handle(k).YData = yFac*asinh(handle(k).YData/yScale);
158          ylimk=[min(handle(k).YData(:)) max(handle(k).YData(:))];
159          ylim1=[min(ylimk(1),ylim1(1)) max(ylimk(2),ylim1(2))];
160      end
161      if ~isinf(zScale)
162          handle(k).ZData = zFac*asinh(handle(k).ZData/zScale);
163          zlimk=[min(handle(k).ZData(:)) max(handle(k).ZData(:))];
164          zlim1=[min(zlimk(1),zlim1(1)) max(zlimk(2),zlim1(2))];
165      end
166      handle(k).UserData = 'quasiLogAxes';
167  end%for k
168 end%for kk

```

Set 4% padding around all margins of transformed data—crude but serviceable. Unless the axis had already been manually set, in which case use the transformed set limits.

```

177  if ~isinf(xScale),
178      if xlim('mode')=="manual"
179          xlim1=xFac*asinh(xlim0/xScale);
180      else xlim1=xlim1+0.04*diff(xlim1)*[-1 1];
181      end, end
182  if ~isinf(yScale),
183      if ylim('mode')=="manual"
184          ylim1=yFac*asinh(ylim0/yScale);
185      else ylim1=ylim1+0.04*diff(ylim1)*[-1 1];
186      end, end
187  if ~isinf(zScale),
188      if zlim('mode')=="manual"
189          zlim1=zFac*asinh(zlim0/zScale);
190      else zlim1=zlim1+0.04*diff(zlim1)*[-1 1];
191      end, end
192  if ~isinf(cScale),
193      if theAxes.CLimMode=="manual"
194          clim1=cFac*asinh(clim0/cScale);
195      else clim1=clim1+    0*diff(clim1)*[-1 1];
196      end, end

```

### Scale axes, and tick marks on axes

```

204 if ~isinf(xScale)
205     xlim(xlim1);
206     tickingQuasiLogAxes(theAxes,'X',xlim1,xScale,xFac)
207 end%if
208 if ~isinf(yScale)
209     ylim(ylim1);
210     tickingQuasiLogAxes(theAxes,'Y',ylim1,yScale,yFac)
211 end%if
212 if ~isinf(zScale)
213     zlim(zlim1);
214     tickingQuasiLogAxes(theAxes,'Z',zlim1,zScale,zFac)
215 end%if

```

But for color, only tick when we find a colorbar.

```

221 if ~isinf(cScale)
222     caxis(clim1);
223 for p=1:numel(theFig.Children)
224     ca = theFig.Children(p);
225     if class(ca) == "matlab.graphics.illustration.ColorBar"
226         tickingQuasiLogAxes(ca,'C',clim1,cScale,cFac)
227         break
228     end
229 end
230 end%if

```

Turn the grid on by default.

```

236 grid on
237 end%function

```

#### 3.20.1 tickingQuasiLogAxes(): typeset ticks and labels on an axis

```
246 function tickingQuasiLogAxes(ca,Q,qlim1,qScale,qFac)
```

##### Input

- **ca:** pointer to axes/colorbar dataset.
- **Q:** character, either X,Y,Z,C.
- **qlim1:** the scaled limits of the axis.
- **qScale:** the scaling parameter for the axis.
- **qFac:** the scaling factor for the axis.

**Output** None, just the ticked and labelled axes.

Get the order of magnitude of the horizontal data.

```

263 qmax=max(abs(qlim1));
264 qmag=floor(log10(qScale*sinh(qmax/qFac)));

```

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

```

272      ticks=10.^(-qmag+(-7:0));
273      j=find(ticks>qScale*sinh(0.06*qmax/qFac));
274      nj=length(j);
275      if nj<3,      ticks=[1;2;5]*ticks(j);
276      elseif nj<5, ticks=[1;3]*ticks(j);
277      else          ticks=ticks(j);
278      end
279      ticks=sort([0;ticks(:);-ticks(:)]);

```

Set the ticks in place according to the transformation.

```

285      if Q=='C', p='s'; Q=''; else p=''; end
286      set(ca,[Q 'Tick' p],qFac*asinh(ticks/qScale) ...
287          ,[Q 'TickLabel' p],cellstr(num2str(ticks,4)))
288      if Q=='X', set(ca,[Q 'TickLabelRotation'],40), end
289  end%function qScaling

```

### 3.21 patchEdgeIntCore(): sets multi-D patch face values from 1D macroscale interpolation in one direction

Couples multi-D patches across multi-D 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 used by `patchEdgeInt1()`, `patchEdgeInt2()` and `patchEdgeInt3()`.

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

```
26 function u = patchEdgeIntCore(l,u,q,patches,stagShift ...
27           ,n1,nl,n2,nEnsem,N1,Nl,N2,lLo,lHi)
```

#### Input

- `l` the direction in space of this interpolation.
- `u` is a vector/array with number of elements  $n_1 \cdot (n_l - 2) \cdot n_2 \cdot n_{Ensem} \cdot N_1 \cdot N_l \cdot N_2$ .
- `q` spatial coordinates in the direction with number of elements  $n_l \cdot N_l$ .
- `patches` a struct set by `configPatches().`
- Get `stagShift` from `patches.??`
- ...

#### Output

- `u` is 7D array,  $n_1 \cdot n_l \cdot n_2 \cdot n_{Ensem} \cdot N_1 \cdot N_l \cdot N_2$ , 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.

```
76 if max(abs(imag(u(:))))<1e-9*max(abs(u(:)))
77     uclean=@(u) real(u);
78 else uclean=@(u) u;
79 end
```

Reshape the field values accordingly, fattened for the new edge values, reshape the coordinate values, and set macroscale indices and periodic neighbours.

```
88 u = cat(2,nan(n1, 1 ,n2,nEnsem,N1,Nl,N2) ...
89   ,reshape(u,n1,nl-2,n2,nEnsem,N1,Nl,N2) ...
90   ,nan(n1, 1 ,n2,nEnsem,N1,Nl,N2) );
91 q = reshape(q,1,nl,1,1,1,Nl,1);
92 Il=1:Nl; Ip=mod(Il,Nl)+1; Im=mod(Il-2,Nl)+1;
93 l0 = round((nl+1)/2);
```

### 3.21.1 Periodic macroscale interpolation schemes

First get the size ratios of the patches in the direction.

```
103 if patches.periodic(1)
104 rl = patches.ratio(1);
```

#### 3.21.1.1 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.

```
114 ordCC = patches.ordCC;
115 if ordCC>0 % then finite-width polynomial interpolation
```

***l*-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.

```
127 if patches.EdgyInt % interpolate next-to-face values
128   U = u(:, [2 nl-1], :, :, :, :, :);
129 else % interpolate centre-cross values
130   U = u(:, 10, :, :, :, :, :);
131 end;%if patches.EdgyInt
```

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

```
139 szU0=size(U); szU0=[szU0 ones(1,7-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.

```
149 if ~patches.parallel, dmu = zeros(szU0); % 8D
150 else dmu = zeros(szU0, patches.codist); % 8D
151 end%if patches.parallel
```

First compute differences  $\mu\delta$  and  $\delta^2$ .

```
157 %disp('starting standard interpolation')
158 dmu(:,:, :, :, :, Il, :, 1) = (U(:,:, :, :, :, Ip, :) ...
159                                -U(:,:, :, :, :, Im, :))/2; \%mu\delta
160 dmu(:,:, :, :, :, Il, :, 2) = (U(:,:, :, :, :, Ip, :) ...
161                                -2*U(:,:, :, :, :, Il, :)) +U(:,:, :, :, :, Im, :)); \%delta^2
```

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

```
168 for k = 3:ordCC
169   dmu(:,:, :, :, :, Il, :, k) =      dmu(:,:, :, :, :, Ip, :, k-2) ...
170   -2*dmu(:,:, :, :, :, Il, :, k-2) +dmu(:,:, :, :, :, Im, :, k-2);
171 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 in `lLo` and `lHi` (from `patches.le`, `patches.ri`, `patches.bo`, `patches.to`, `patches.ba`, and `patches.fr`).

```

188 k=1+patches.EdgyInt; % use centre or two faces
189 u(:,nl,:,:lHi,:,:,:) ...
190 = U(:,1,:,:,:,:,*)*(1-patches.stag) ...
191 +sum( shiftdim(patches.Cwtsr(:,1),-7).*dmu(:,1,:,:,:, :, :, :, :),8);
192 u(:,1 ,:,lLo,:,:,:) ...
193 = U(:,k,:,:,:,:,*)*(1-patches.stag) ...
194 +sum( shiftdim(patches.Cwtsl(:,1),-7).*dmu(:,k,:,:,:, :, :, :, :),8);
```

### 3.21.1.2 Case of spectral interpolation

Assumes the domain is macro-periodic.

```
205 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  $N$ .

**$l$ -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,  $k = (0, 1, \dots, k_{\max}, +(k_{\max} + 1) - k_{\max}, \dots, -1)$ .

```

229 kMax = floor((Nl-1)/2);
230 kr = shiftdim( rl*2*pi/Nl*(mod((0:Nl-1)+kMax,Nl)-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 `lLo` and `lHi` (from `patches.le`, `patches.ri`, `patches.to`, `patches.bo`, `patches.fr` and `patches.ba`).

```

245 if ~patches.EdgyInt
246     Cm = fft( u(:,l0,:,:,:, :, :, :),[],6);
247     Cp = Cm;
248 else
249     Cm = fft( u(:, 2,:,:lLo,:,:,:) ,[],6);
```

```

250      Cp = fft( u(:,nl-1,:,:lHi,:,:,:) ,[],6);
251  end%if ~patches.EdgyInt

```

Now invert the Fourier transforms to complete interpolation. Enforce reality when appropriate with `uclean()`.

```

258  u(:,nl,:,:,:, :,:) = uclean( ifft( ...
259      Cm.*exp(1i*(stagShift+kr)) ,[],6) );
260  u(:, 1,:,:,:, :,:) = uclean( ifft( ...
261      Cp.*exp(1i*(stagShift-kr)) ,[],6) );

```

End the two periodic cases of spectral or finite-difference interpolation.

```
269  end% if ordCC>0
```

### 3.21.2 Non-periodic macroscale interpolation

```

278  else% patches.periodic false
279  assert(~patches.stag, ...
280  'not implementing staggered grids for non-periodic')

```

Determine the order of interpolation `pl` (potentially different in the different directions!), and hence size of the (forward) divided difference tables in `F` (8D) 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.

```

293  if patches.ordCC<1, pl = Nl-1;
294  else pl = min(patches.ordCC,Nl-1);
295  end

```

#### 3.21.2.1 $l$ -direction values

Set function values in second 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. <sup>12</sup>

```

308  F = nan(n1,patches.EdgyInt+1,n2,nEnsem,N1,Nl,N2,pl+1);
309  if patches.EdgyInt % interpolate next-to-face values
310      F(:,:, :, :, :, :, 1) = u(:, [nl-1 2], :, :, :, :, :);
311      Q = q(:, [nl-1 2], :, :, :, :, :);
312  else % interpolate mid-patch cross-patch values
313      F(:,:, :, :, :, :, 1) = u(:, 10, :, :, :, :, :);
314      Q = q(:, 10, :, :, :, :, :);
315  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,

<sup>12</sup> **ToDo:** Have no plans to implement core averaging as yet.

and front/back). Recursively find all divided differences in the respective direction.

```

328  for m = 1:pl
329    i = 1:Nl-m;
330    F(:,:,,:,i,:,:,m+1) ...
331    = ( F(:,:,,:,i+1,:,:,m)-F(:,:,,:,i,:,:,m)) ...
332      ./ (Q(:,:,,:,i+1,:,:) - Q(:,:,,:,i,:));
333  end

```

**Interpolate with divided differences** Now interpolate to find the face-values on left/right faces at `Qface` for every interior in other coordinates.

```
342  Qface = q(:,[1 nl],:,:,:,:);
```

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.

```

353  i = max(1,min(1:Nl,Nl-ceil(pl/2))-floor(pl/2));
354  Uface = F(:,:,i,:,:,pl+1);
355  for m = pl:-1:1
356    Uface = F(:,:,i,:,:,m) ...
357    +(Qface-Q(:,:,i,:,:,i+m-1,:)).*Uface;
358  end

```

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

```

366  u(:,1 ,:,lLo,:,:, :) = Uface(:,1 ,:,lLo,:,:, :);
367  u(:,nl,:,lHi,:,:, :) = Uface(:,2 ,:,lHi,:,:, :);

```

### 3.21.2.2 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, unless testing, override their computed interpolation values with `NaN`.

```

381  if isfield(patches,'intTest')&&patches.intTest
382  else %disp('usual case requires user to set bdry values')
383    u(:, 1 ,:, :, 1 ,:) = nan;
384    u(:,nl ,:, :, Nl ,:) = nan;
385  end%if

```

End of the non-periodic interpolation code.

```
392  end%if patches.periodic else
```

Fin, returning the 7D array of field values with interpolated face.

```
401  end% function patchEdgeIntCore
```

### 3.21.3 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, and the number of realisations of cases.

```
22 clear all, close all
23 global patches
24 nRealise = 20
```

#### 3.21.3.1 Check divided difference interpolation

```
36 fprintf('\n\n**** Check divided difference interpolation\n')
37 pause(1)
```

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.)

```
47 for iReal=1:nRealise
48     nEdge = randi(3)% =1,2, or 3
49     edgyInt = rand<0.5
50     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
51     ordCC = 2*randi(4)
52     nPatch = ordCC+randi([2 4])
53     Domain=5*[-rand rand]
54     dx=rand*diff(Domain)/nPatch/nSubP
55     configPatches1(@sin,Domain,'equispace',nPatch,ordCC,dx,nSubP ...
56         , 'EdgyInt',edgyInt,'nEdge',nEdge);
57     patches.intTest = true;
```

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;
```

Copy data, and set edges to `inf` so we can be certain that interpolation is computing the required edge values.

```
85 u=u0; u([1:nEdge end-nEdge+1:end],:)=inf;
```

Then evaluate the interpolation and squeeze the singleton dimension of an ‘ensemble’.

```
92     ui=patchEdgeInt1(u(:));
93     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.

```
103    j=1:nPatch;
104    iError=ui(:,:,j)-u0(:,:,j);
105    hist(log10(abs(iError(abs(iError)>0))),-17:-9)
106    xlabel('log10 iError'), pause(0.3)%%
107    normError=norm(iError(:))
108    assert(normError<1e-13*4^ordCC ...
109    , 'failed divided difference interpolation')
```

End the for-loop over random parameters.

```
116 end%for iReal
117 fprintf('\n\nPassed all divided difference interpolation\n')
```

### 3.21.3.2 Test standard spectral interpolation

```
133 fprintf('\n\n**** Test standard spectral interpolation\n')
134 pause(1)
```

Test over random numbers of patches, random domain lengths, random microscale spacing, random choice of `edgyInt`. Say do fifteen realisations.

```
142 for iReal=1:nRealise
143     nEdge=randi(3)% =1,2, or 3
144     edgyInt = rand<0.5
145     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
146     nPatch=randi([5 10])
147     Len=10*rand
148     dx=0.5*rand*Len/nPatch/nSubP
149     configPatches1(@sin,[0 Len], 'periodic',nPatch,0,dx,nSubP ...
150         , 'EdgyInt',edgyInt,'nEdge',nEdge); % random Edgy or not
151     if mod(nPatch,2)==0, fprintf('\nAvoiding highest wavenumber\n'), end
152     kMax=floor((nPatch-1)/2);
```

**Test single field** Set a profile, and evaluate the interpolation.

```
160 for k=-kMax:kMax
161     u0=exp(1i*k*patches.x*2*pi/Len);
162     u=u0; u([1:nEdge end-nEdge+1:end],:)=nan;
163     ui=patchEdgeInt1(u(:));
164     normError=rms(ui(:)-u0(:));
165     if abs(normError)>5e-14
```

```

166     normError=normError, k=k
167     error(['failed single var interpolation k=' num2str(k)])
168   end
169 end

```

**Test multiple fields** Use this to measure some of the errors in order to omit singleton dimensions,

```

177 normDiff=@(u,v) ...
178 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.

```

188 for k=1:(nPatch-1)/2 % not checking the highest wavenumber
189   u0=sin(k*patches.x*2*pi/Len);
190   v0=cos(k*patches.x*2*pi/Len);
191   uvi=patchEdgeInt1( reshape([u0 v0],[],1) );
192   normuError=normDiff(uvi(:,1,:,:),u0);
193   normvError=normDiff(uvi(:,2,:,:),v0);
194   if abs(normuError)+abs(normvError)>2e-13
195     normuError=normuError, normvError=normvError
196     error(['failed double field interpolation k=' num2str(k)])
197   end
198 end

```

End the for-loop over various geometries.

```

205 end
206 fprintf('\nPassed standard spectral interpolation tests\n')

```

### 3.21.3.3 Now test spectral interpolation on staggered grid

```

221 fprintf('\n\n**** Test spectral interpolation on staggered\n')
222 disp('**** Actually not doing so as DNW following changes Oct 2023')
223 pause(1)

```

Must have even number of patches for a staggered grid. Have not yet implemented multiple edge values for a staggered grid as I am uncertain whether it makes any sense—certainly this test fails anyway.

```

232 for iReal=1:0%?? nRealise
233   nEdge = 1 % required
234   edgyInt = rand<0.5
235   nPatch=2*randi([3 10])
236   nSubP=7 % of form 4*N-1
237   Len=10*rand
238   dx=0.5*rand*Len/nPatch/nSubP
239   configPatches1(@simpleWavePde,[0 Len],'periodic' ...
240                 ,nPatch,-1,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

```

241      if mod(nPatch,4)==0, fprintf('\nAvoiding highest wavenumber\n'), end
242      kMax=floor((nPatch/2-1)/2)

```

Identify which microscale grid points are  $h$  or  $u$  values.

```

248 uPts=mod( (1:nSubP)+(1:nPatch) ,2);
249 hPts=find(1-uPts);
250 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.

```

258 fprintf('Staggered: single field-pair test.\n')
259 for k=-kMax:kMax
260   U0=nan(nSubP,nPatch);
261   U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
262   U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
263   U=U0;
264   U([1:nEdge end-nEdge+1:end],:)=nan;
265   Ui=patchEdgeInt1(U0(:));
266   normError=norm(Ui(:)-U0(:));
267   if abs(normError)>5e-14
268     normError=normError
269     patches=patches
270     error(['staggered: failed single sys interpolation k=' num2str(k)])
271   end
272 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).

```

283 normDiff=@(u,v,w) ...
284 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.

```

294 fprintf('Staggered: Two field-pairs test.\n')
295 x0=patches.x((nSubP+1)/2,1);
296 patches.x=patches.x-x0;
297 oddP=1:2:nPatch; evnP=2:2:nPatch;
298 for k=1:kMax
299   U0=nan(nSubP,1,1,nPatch); V0=U0;
300   U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
301   U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
302   U=U0; U([1:nEdge end-nEdge+1:end],:)=nan;
303   V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
304   V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
305   V=V0; V([1:nEdge end-nEdge+1:end],:)=nan;

```

```

306    UVi=patchEdgeInt1([U0 V0]);
307    normuError=[normDiff(UVi(:,1,:,:oddP),U0(:, :, :, oddP),U0(:, :, :, evnP))
308                normDiff(UVi(:,1,:,:evnP),U0(:, :, :, evnP),U0(:, :, :, oddP))]';
309    normvError=[normDiff(UVi(:,2,:,:oddP),V0(:, :, :, oddP),V0(:, :, :, evnP))
310                normDiff(UVi(:,2,:,:evnP),V0(:, :, :, evnP),V0(:, :, :, oddP))]';
311    if norm(normuError)+norm(normvError)>2e-13
312        normuError=normuError, normvError=normvError
313        patches=patches
314        error(['staggered: failed double field interpolation k=' num2str(k)])
315    end
316 end
317
318     End for-loop over patches
319
320 end

```

### 3.21.3.4 Check standard finite width interpolation

```

337 fprintf('\n\n**** Check standard finite width interpolation\n')
338 pause(1)

```

Check over various types and orders of interpolation, numbers of patches, random domain lengths and random ratios. (The @sin is a dummy.)

```

346 for iReal=1:nRealise
347     nEdge=randi(3)% =1,2, or 3
348     edgyInt = rand<0.5
349     nSubP = nEdge*( (2-edgyInt)*randi(2)+1+edgyInt )
350     ordCC = 2*randi(4)
351     nPatch = ordCC+randi([2 4])
352     Domain=5*[-rand rand]
353     dx=0.5*rand*diff(Domain)/nPatch/nSubP
354     configPatches1(@sin,Domain,'periodic',nPatch,ordCC,dx,nSubP ...
355     , 'EdgyInt',edgyInt,'nEdge',nEdge);

```

**Check multiple fields simultaneously** Set profiles to be various powers of  $x$ ,  $ps$ , and store as different ‘variables’ at each point.

```

364     ps=1:ordCC
365     cs=randn(size(ps));
366     u0=patches.x.^ps.*cs+randn;

```

Copy data, and set edges to NaN so we can be certain that interpolation is computing the required edge values.

```

373     u=u0; u([1:nEdge end-nEdge+1:end],:)=nan;

```

Then evaluate the interpolation and squeeze the singleton dimension of an ‘ensemble’.

```

380     ui=patchEdgeInt1(u(:));
381     ui=squeeze(ui);

```

The interior patches should have zero error.

```

387      j=ordCC/2+1:nPatch-ordCC/2;
388      iError=ui(:,:,j)-u0(:,:,j);
389      normError=norm(iError(:))
390      assert(normError<5e-12 ...
391      ,’failed finite stencil interpolation’)

      End the for-loops over various parameters.

398  end%for iReal
399  fprintf(’\nPassed all standard polynomial interpolation\n’)

```

### 3.21.3.5 Now test finite width interpolation on staggered grid

```

414  fprintf(’\n\n**** Check finite width staggered\n’)
415  disp(’**** Actually not doing so as DNW following changes Oct 2023’)
416  pause(1)

```

Must have even number of patches for a staggered grid.

```

422  for iReal=1:0%?? nRealise
423      nEdge = 1 % required for now
424      edgyInt = rand<0.5
425      nPatch=2*randi([3 10])
426      nSubP=3; % of form 4*N-1
427      Len=10*rand
428      dx=0.5*rand*Len/nPatch/nSubP
429      configPatches1(@simpleWavePde,[0 Len],’periodic’ ...
430      ,nPatch,-1,dx,nSubP,’EdgyInt’,edgyInt,’nEdge’,nEdge);
431      kMax=floor((nPatch/2-1)/2)

```

Identify which microscale grid points are  $h$  or  $u$  values.

```

438  uPts=mod( (1:nSubP)+(1:nPatch) ,2);
439  hPts=find(1-uPts);
440  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.

```

448  fprintf(’\nSingle field-pair test.\n’)
449  for k=-kMax:kMax
450      U0=nan(nSubP,nPatch);
451      U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
452      U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
453      Ui=squeeze(patchEdgeInt1(U0(:)));
454      normError=norm(Ui-U0);
455      if abs(normError)>5e-14
456          normError=normError
457          error(['failed single sys interpolation k=' num2str(k)])
458      end
459  end

```

**Test multiple fields** 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.

```

471 i0=(nSubP+1)/2; % centre-patch index
472 fprintf('Two field-pairs test.\n')
473 x0=patches.x((nSubP+1)/2,1);
474 patches.x=patches.x-x0;
475 for k=1:nPatch/4
476     U0=nan(nSubP,1,1,nPatch); V0=U0;
477     U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
478     U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
479     V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
480     V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
481     UVi=patchEdgeInt1([U0 V0]);
482     Ui=squeeze(UVi(:,1,1,:));
483     Vi=squeeze(UVi(:,2,1,:));
484     normuError=norm(Ui(:,1:2:nPatch)-U0(:,1:2:nPatch))*norm(U0(i0,2:2:nPatch))
485         +norm(Ui(:,2:2:nPatch)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPatch));
486     normvError=norm(Vi(:,1:2:nPatch)-V0(:,1:2:nPatch))*norm(V0(i0,2:2:nPatch))
487         +norm(Vi(:,2:2:nPatch)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPatch));
488     if abs(normuError)+abs(normvError)>2e-13
489         normuError=normuError, normvError=normvError
490         error(['failed double field interpolation k=' num2str(k)])
491     end
492 end
493
End for-loop over the realisations
494
495 end

```

### 3.21.3.6 Finish

If no error messages, then all OK.

```
511 fprintf('\n**** If you read this, then all tests were passed\n')
```

---

## 4 Patches in 2D space

---

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

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

```
19 function patches = configPatches2(fun,Xlim,Dom ...
20     ,nPatch,ordCC,dx,nSubP,varargin)
21 version = '2023-10-19';
```

**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.

But, if `nPatch(n) = 1`, then in the spatial direction `n` the domain is the mean of `Xlim(2*n-1:2*n)` plus/minus half the patch-width in that direction.

- `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` determines the number of spatial patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches (an integer  $\geq 1$ ) 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—adjusted a little when `nEdge > 1`. 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 both directions, otherwise `nSubP(1:2)` gives the number in each direction. If not using `EdgyInt`, then `nSubP./nEdge` must be odd integer(s) so that there is/are centre-patch lattice lines. So for the defaults of `nEdge = 1` and not `EdgyInt`, then `nSubP` must be odd.
- ‘`nEdge`’, *optional* (integer—scalar or two element), default=1, the width of edge values set by interpolation at the edge regions of each patch. If two elements, then respectively the width in  $x, y$ -directions. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre cross-patch lines.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- **hetCoeffs**, *optional*, default empty. Supply a 2D or 3D array of microscale heterogeneous coefficients to be used by the given microscale **fun** in each patch. Say the given array **cs** is of size  $m_x \times m_y \times n_c$ , where  $n_c$  is the number of different sets of coefficients. For example, in heterogeneous diffusion,  $n_c = 2$  for the diffusivities in the *two* different spatial directions (or  $n_c = 3$  for the diffusivity tensor). The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the  $n_c$  coefficients being  $n_c$  parameters in some macroscale formula.
    - If **nEnsem** = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the (1, 1)-point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of **nSubP**-2\***nEdge** for **EdgyInt**, or a factor of (**nSubP**-**nEdge**)/2 for not **EdgyInt**.
    - If **nEnsem** > 1 (value immaterial), then reset **nEnsem** :=  $m_x \cdot m_y$  and construct an ensemble of all  $m_x \cdot m_y$  phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When **EdgyInt** is true, and when the coefficients are diffusivities/elasticities in  $x$  and  $y$  directions, respectively, then this coupling cunningly preserves symmetry.
  - **'parallel'**, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.
- If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then it will distribute the patches over multiple CPUs/cores. In MATLAB, only one array dimension can be split in the distribution, so it chooses the one space dimension  $x, y$  corresponding to the highest **\nPatch** (if a tie, then chooses the rightmost of  $x, y$ ). A user may correspondingly distribute arrays with property **patches.codist**, or simply use formulas invoking the preset distributed arrays **patches.x**, and **patches.y**. If a user has not yet established a parallel pool, then a 'local' pool is started.

**Output** The struct **patches** is created and set with the following components. If no output variable is provided for **patches**, then make the struct available as a global variable.<sup>1</sup>

```
223 if nargout==0, global patches, end
224 patches.version = version;
```

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

---

<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.

- `.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` 1 × 2, is the width of edge values set by interpolation at the edge regions of each patch, in the  $x, y$ -directions respectively.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - [] 0D, or
  - if `nEnsem` = 1,  $(nSubP(1) - 1) \times (nSubP(2) - 1) \times n_c$  3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1,  $(nSubP(1) - 1) \times (nSubP(2) - 1) \times n_c \times m_x m_y$  4D array of  $m_x m_y$  ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

#### 4.1.1 If no arguments, then execute an example

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

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

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

### 3. process results

Establish global patch data struct to interface with a function coding a nonlinear ‘diffusion’ PDE: to be solved on  $6 \times 4$ -periodic domain, with  $9 \times 7$  patches, spectral interpolation (0) couples the patches, with  $5 \times 5$  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).

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

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

```
341 u0 = exp(-patches.x.^2-patches.y.^2);
342 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.

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

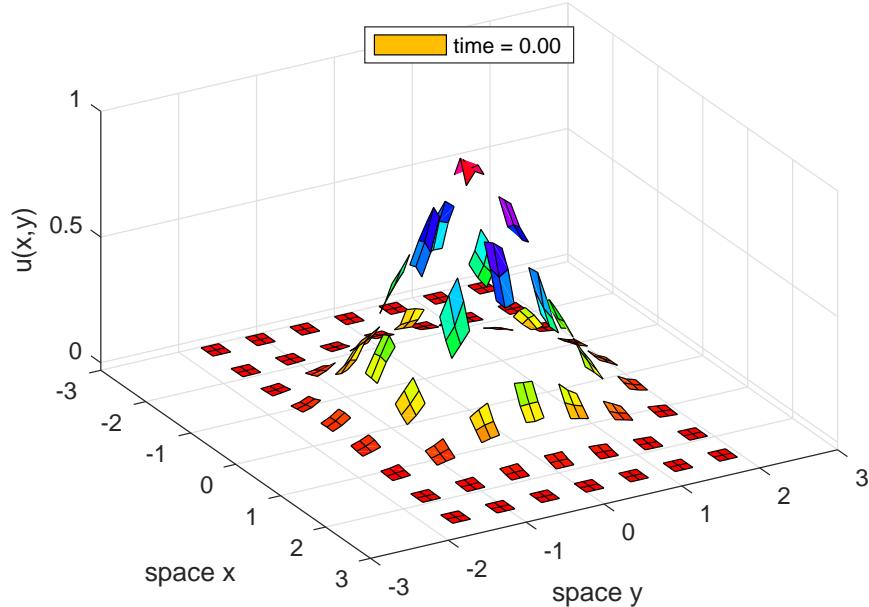
Start by showing the initial conditions of [Figure 4.1](#) while the simulation computes.

```
359 u = reshape(permute(squeeze(u0) ...
360 , [1 3 2 4]), [numel(x) numel(y)]);
361 hsurf = mesh(x(:, ), y(:, ), u');
362 axis([-3 3 -3 3 -0.03 1]), view(60, 40)
363 legend('time = 0.00', 'Location', 'north')
364 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y)$')
365 colormap(hsv)
366 if !ourCf2eps([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.

```
383 disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
384 drawnow
385 if ~exist('OCTAVE_VERSION', 'builtin')
386     [ts, us] = ode23(@patchSys2, linspace(0, 2).^2, u0(:));
387 else % octave version is quite slow for me
388     lsode_options('absolute tolerance', 1e-4);
389     lsode_options('relative tolerance', 1e-4);
390     [ts, us] = odeOctave(@patchSys2, [0 1], u0(:));
391 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.

```

399 for i = 1:length(ts)
400     u = patchEdgeInt2(us(i,:));
401     u = reshape(permute(squeeze(u) ...
402                 ,[1 3 2 4]), [numel(x) numel(y)]);
403     set(hsurf,'ZData', u');
404     legend(['time = ' num2str(ts(i),'%4.2f')])
405     pause(0.1)
406 end
407 ifOrCf2eps([mfilename 't3'])

```

Upon finishing execution of the example, exit this function.

```

422 return
423 end%if no arguments

```

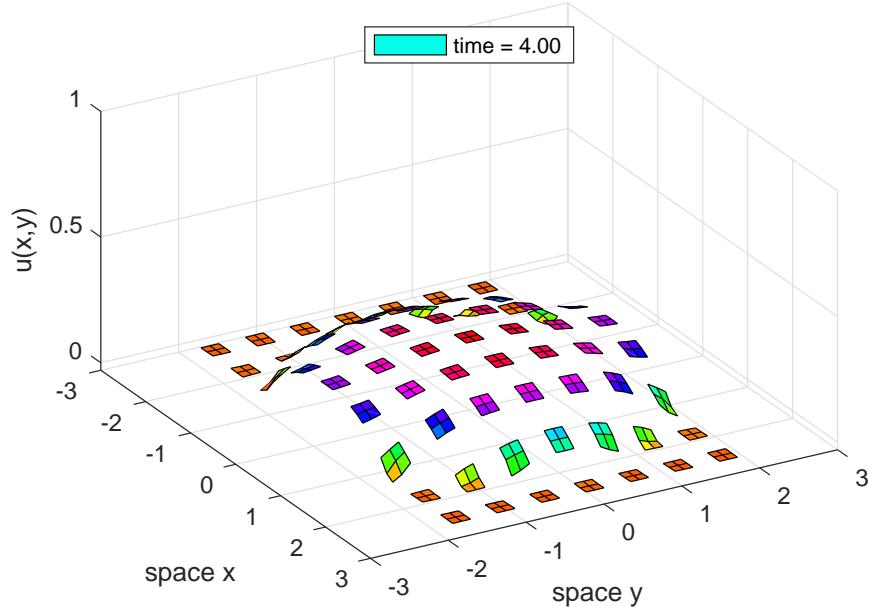
**Example of nonlinear diffusion PDE inside patches** As a microscale discretisation of  $u_t = \nabla^2(u^3)$ , code  $\dot{u}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i-1,j,k,l}^3) + \frac{1}{\delta y^2}(u_{i,j+1,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i,j-1,k,l}^3)$ .

```

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

```

437 p = inputParser;
438 fnValidation = @(f) isa(f, 'function_handle');%test for fn name
439 addRequired(p, 'fun', fnValidation);
440 addRequired(p, 'Xlim', @isnumeric);
441 %addRequired(p, 'Dom'); % nothing yet decided
442 addRequired(p, 'nPatch', @isnumeric);
443 addRequired(p, 'ordCC', @isnumeric);
444 addRequired(p, 'dx', @isnumeric);
445 addRequired(p, 'nSubP', @isnumeric);
446 addParameter(p, 'nEdge', 1, @isnumeric);
447 addParameter(p, 'EdgyInt', false, @islogical);
448 addParameter(p, 'nEnsem', 1, @isnumeric);
449 addParameter(p, 'hetCoeffs', [], @isnumeric);
450 addParameter(p, 'parallel', false, @islogical);
451 %addParameter(p, 'nCore', 1, @isnumeric); % not yet implemented
452 parse(p, fun, Xlim, nPatch, ordCC, dx, nSubP, varargin{:});

```

Set the optional parameters.

```

458 patches.nEdge = p.Results.nEdge;
459 if numel(patches.nEdge)==1
460     patches.nEdge = repmat(patches.nEdge, 1, 2);
461 end

```

```

462 patches.EdgyInt = p.Results.EdgyInt;
463 patches.nEnsem = p.Results.nEnsem;
464 cs = p.Results.hetCoeffs;
465 patches.parallel = p.Results.parallel;
466 %patches.nCore = p.Results.nCore;

Initially duplicate parameters for both space dimensions as needed.

474 if numel(Xlim)==2, Xlim = repmat(Xlim,1,2); end
475 if numel(nPatch)==1, nPatch = repmat(nPatch,1,2); end
476 if numel(dx)==1, dx = repmat(dx,1,2); end
477 if numel(nSubP)==1, nSubP = repmat(nSubP,1,2); end

Check parameters.

484 assert(all(nPatch==round(nPatch))&all(nPatch>0) ...
485     , 'nPatch must be integer(s) and at least one')
486 assert(Xlim(1)<Xlim(2) ...
487     , 'first pair of Xlim must be ordered increasing')
488 assert(Xlim(3)<Xlim(4) ...
489     , 'second pair of Xlim must be ordered increasing')
490 assert((mod(ordCC,2)==0)|all(patches.nEdge==1) ...
491     , 'Cannot yet have nEdge>1 and staggered patch grids')
492 assert(all(3*patches.nEdge<=nSubP) ...
493     , 'too many edge values requested')
494 assert(all(rem(nSubP,patches.nEdge)==0) ...
495     , 'nSubP must be integer multiple of nEdge')
496 if ~patches.EdgyInt, assert(all(rem(nSubP./patches.nEdge,2)==1) ...
497     , 'for non-edgyInt, nSubP./nEdge must be odd integer')
498 end
499 if (patches.nEnsem>1)&all(patches.nEdge>1)
500     warning('not yet tested when both nEnsem and nEdge non-one')
501 end
502 %if patches.nCore>1
503 %    warning('nCore>1 not yet tested in this version')
504 %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.

```

515 if ~isstruct(Dom), pre2023=isnan(Dom);
516 else pre2023=false; end
517 if pre2023, ratio=dx; dx=nan; end

```

Default macroscale conditions are periodic with evenly spaced patches.

```

525 if isempty(Dom), Dom=struct('type','periodic'); end
526 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.

```

534 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.

```
547 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.

```
556 patches.periodic=false(2,1);
557 for p=1:2
558 switch Dom.type(p,:)
559 case {'periodic','periodic'}
560     patches.periodic(p) = true;
561     msg=' not available for Dom.type = periodic';
562     if isfield(Dom,'bcOffset')&&~all(isnan(Dom.bcOffset(:,p)))
563         warning(['bcOffset' msg]), end
564     if isfield(Dom,'X'), warning(['X' msg]), end
565     if isfield(Dom,'Y'), warning(['Y' msg]), end
566 case {'equispace','chebyshev'}
567     if ~isfield(Dom,'bcOffset')
568         Dom.bcOffset=nan(2,2); Dom.bcOffset(:,p)=0;
569     else Dom.bcOffset( isnan(Dom.bcOffset(:,p)) ,p)=0;
570     end
571 % for mixed with usergiven, following should still work
572 if numel(Dom.bcOffset)==1
573     Dom.bcOffset=repmat(Dom.bcOffset,2,2); end
574 if numel(Dom.bcOffset)==2
575     Dom.bcOffset=repmat(Dom.bcOffset(:,2,1); end
576     msg=' not available for Dom.type = equispace or chebyshev';
577     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
578     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
579 case 'usergiven'
580 %     if isfield(Dom,'bcOffset')
581 %         warning('bcOffset not available for usergiven Dom.type'), end
582     msg=' required for Dom.type = usergiven';
583     if p==1, assert(isfield(Dom,'X'),['X' msg]), end
584     if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
585 otherwise
586     error([Dom.type(p,:) ' is unknown Dom.type'])
587 end%switch Dom.type
588 end%for p
```

#### 4.1.3 The code to make patches

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

---

```
601 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. <sup>2</sup>

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

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

```
619 patches.stag = mod(ordCC,2);
620 assert(patches.stag==0,'staggered not yet implemented??')
621 ordCC = ordCC+patches.stag;
622 patches.ordCC = ordCC;
```

Check for staggered grid and periodic case.

```
628 if patches.stag, assert(all(mod(nPatch,2)==0), ...
629     'Require an even number of patches for staggered grid')
630 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.

```
643 patches.ratio = nan(1,2);
644 for q=1:2
645 qq=2*q-1; qstr=num2str(q);
```

Distribution depends upon `Dom.type`:

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

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

```
659 case {'periodic','periodic'}
660 Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
661 DQ=Q(2)-Q(1);
662 Q=Q(1:nPatch(q))+diff(Q)/2;
663 pEI=patches.EdgyInt; % abbreviation
664 pnE=patches.nEdge(q);% abbreviation
665 if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-pnE*(1+pEI))*(2-pEI);
666 else      ratio(q) = dx(q)/DQ*(nSubP(q)-pnE*(1+pEI))/(2-pEI);
667 end
668 patches.ratio(q)=ratio(q);
```

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

---

<sup>2</sup> **ToDo:** Perhaps implement staggered spectral coupling.

```

677 case 'equispace'
678     halfWidth=dx(q)*(nSubP(q)-1)/2;
679     if nPatch(q)>1
680         Q=linspace(Xlim(qq)+halfWidth-Dom.bcOffset(qq)*dx(q) ...
681                     ,Xlim(qq+1)-halfWidth+Dom.bcOffset(qq+1)*dx(q) ...
682                     ,nPatch(q));
683         DQ=diff(Q(1:2));
684     else% nPatch(q)==1
685         warning(['nPatch(' qstr ')==1 so placing one patch at the mean of Xlim']);
686         Q=mean(Xlim(qq:qq+1));
687         DQ=Inf;
688     end%if nPatch(q)
689     width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
690     if DQ<width*0.999999
691         warning(['excess equispace patches (double overlap) in dirn ' qstr])
692     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’.<sup>3</sup>

```

709 case 'chebyshev'
710     halfWidth=dx(q)*(nSubP(q)-1)/2;
711     if nPatch(q)>1
712         Q1 = Xlim(qq) +halfWidth-Dom.bcOffset(qq)*dx(q);
713         Q2 = Xlim(qq+1)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
714         % 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`.

```

723 pEI=patches.EdgyInt; % abbreviation
724 pnE=patches.nEdge(q);% abbreviation
725 width=(1+pEI)/2*(nSubP(q)-pnE*(1+pEI))*dx(q);
726 for b=0:2:nPatch(q)-2
727     DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
728     if DQmin>width, break, end
729 end%for b
730 if DQmin<width*0.999999
731     warning(['excess Chebyshev patches (mid-domain overlap) in dirn ' qstr])
732 end

```

Assign the centre-patch coordinates.

```

738 Q =[ Q1+(0:b/2-1)*width ...
739             (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...

```

---

<sup>3</sup> 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??

```

740      Q2+(1-b/2:0)*width ];
741  else% nPatch(q)==1
742    warning(['nPatch(' qstr ')==1 so placing one patch at the mean of Xlim']);
743    Q=mean(Xlim(qq:qq+1));
744  end%if nPatch(q)

```

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

```

753 case 'usergiven'
754   if q==1, Q = reshape(Dom.X,1,[]);
755   else     Q = reshape(Dom.Y,1,[]);
756   end%if
757 end%switch Dom.type

```

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

```

764 if q==1, X=Q; end
765 if q==2, Y=Q; end
766 end%for q

```

**Construct the micro-grids** Fourth, construct the microscale grid in each patch, centred about the given mid-points  $X, Y$ . Reshape the grid to be 6D to suit dimensions (micro,Vars,Ens,macro).

```

782 xs = dx(1)*( (1:nSubP(1))-mean(1:nSubP(1)) );
783 patches.x = reshape( xs'+X ...
784                           ,nSubP(1),1,1,1,nPatch(1),1);
785 ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
786 patches.y = reshape( ys'+Y ...
787                           ,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.<sup>4</sup>

```

798 if any(patches.periodic)
799 % ratio = reshape(ratio,1,2); % force to be row vector
800 % patches.ratio=ratio;
801 if ordCC>0
802   [Cwtsr,Cwtsl] = patchCwts(ratio,ordCC,patches.stag);
803   patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
804 end%if
805 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

---

<sup>4</sup> **ToDo:** Might sometime extend to coupling via derivative values.

- the left-edge/centre realisations  $1:n_{\text{Ensem}}$  are to interpolate to **re**.

**re** and **li** are ‘transposes’ of each other as  $\text{re}(\text{li})=\text{le}(\text{ri})$  are both  $1:n_{\text{Ensem}}$ . 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()**.

```
832 nE = patches.nEnsem;
833 patches.le = 1:nE; patches.ri = 1:nE;
834 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.

```
846 if ~isempty(cs)
847     [mx,my,nc] = size(cs);
848     nx = nSubP(1); ny = nSubP(2);
849     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.

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

But for  $n_{\text{Ensem}} > 1$  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.

```
866     patches.nEnsem = mx*my;
867     patches.cs = nan(nx-1,ny-1,nc,mx,my);
868     for j = 1:my
869         js = (j:j+ny-2);
870         for i = 1:mx
871             is = (i:i+nx-2);
872             patches.cs(:,:,i,j) = cs(is,js,:);
873         end
874     end
875     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.

```
885 le = mod((0:mx-1)+mod(nx-2,mx),mx)+1;
886 patches.le = reshape( le'+mx*(0:my-1) ,[],1);
887 ri = mod((0:mx-1)-mod(nx-2,mx),mx)+1;
888 patches.ri = reshape( ri'+mx*(0:my-1) ,[],1);
889 bo = mod((0:my-1)+mod(ny-2,my),my)+1;
```

```

890     patches.bo = reshape( (1:mx)' + mx*(bo-1) ,[],1);
891     to = mod((0:my-1)-mod(ny-2,my),my)+1;
892     patches.to = reshape( (1:mx)' + mx*(to-1) ,[],1);

```

Issue warning if the ensemble is likely to be affected by lack of scale separation.

<sup>5</sup>

```

900 if prod(ratio)*patches.nEnsem>0.9, warning( ...
901 'Probably poor scale separation in ensemble of coupled phase-shifts')
902 scaleSeparationParameter = ratio*patches.nEnsem
903 end

End the two if-statements.

909 end%if-else nEnsem>1
910 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>6</sup>

```

929 if patches.parallel
930 % theparpool=gcp()
931 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.

```

940 [~,pari]=max(nPatch+0.01*(1:2));
941 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.

```

951 switch pari
952   case 1, patches.x=codistributed(patches.x,patches.codist);
953   case 2, patches.y=codistributed(patches.y,patches.codist);
954 otherwise
955   error('should never have bad index for parallel distribution')
956 end%switch
957 end%spmd

```

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

---

<sup>5</sup> **ToDo:** Maybe need to justify this and the arbitrary threshold more carefully??

<sup>6</sup> If subsequently outside `spmd`, then one must use functions like `getfield(patches{1}, 'a')`.

```
965 else% not parallel
966     if isfield(patches,'codist'), rmfield(patches,'codist'); end
967 end%if-parallel
```

**Fin**

```
976 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. Communicate patch-design variables (Section 4.1) either via the global struct `patches` or via an optional third argument. `patches` is required for the parallel computing of `spmd`, or if parameters are to be passed though to the user microscale function.

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

### Input

- `u` is a vector/array of length `prod(nSubP) · nVars · nEnsem · prod(nPatch)` where there are `nVars · nEnsem` field values at each of the points in the `nSubP(1) × nSubP(2) × nPatch(1) × nPatch(2)` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches,...)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1) × nSubP(2) × nVars × 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 list of parameters to be passed onto the users time-derivative function as specified in `configPatches2`.

### Output

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

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()`.

```
93 sizeu = size(u);
94 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.

```
104 dudt = patches.fun(t,u,patches,varargin{:});  
105 m = patches.nEdge(1);  
106 dudt([1:m end-m+1:end],:,:) = 0;  
107 m = patches.nEdge(2);  
108 dudt(:,[1:m end-m+1:end],:) = 0;  
109 dudt = reshape(dudt,sizeu);
```

Fin.

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Assumes patch edge values are determined by macroscale interpolation of the patch centre-plane values (Roberts et al. 2014, Bunder et al. 2021), or patch next-to-edge values which appears better (Bunder et al. 2020). This function is primarily used by patchSys3() but is also useful for user graphics.<sup>7</sup>

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 = patchEdgeInt2(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  $\text{nVars} \cdot \text{nEnsem}$  field values at each of the points in the  $\text{nSubP1} \cdot \text{nSubP2} \cdot \text{nPatch1} \cdot \text{nPatch2}$  multiscale spatial grid on the  $\text{nPatch1} \cdot \text{nPatch2}$  array of patches.
- **patches** a struct set by configPatches2() which includes the following information.
  - **.x** is  $\text{nSubP1} \times 1 \times 1 \times 1 \times \text{nPatch1} \times 1$  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  $1 \times \text{nSubP2} \times 1 \times 1 \times 1 \times \text{nPatch2}$  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  $\{0, 2, 4, \dots\}$
  - **.periodic**, logical 2-array, 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.
  - **.Cwtsr** and **.Cwtsl** are the coupling coefficients for finite width interpolation in each of the  $x, y$ -directions—when invoking a periodic direction.
  - **.EdgyInt**, true/false, for determining patch-edge values by interpolation: true, from opposite-edge next-to-edge values (often

---

<sup>7</sup> Script `patchEdgeInt2test.m` verifies most of this code.

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

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

## Output

- `u` is 6D array,  $n_{\text{SubP1}} \cdot n_{\text{SubP2}} \cdot n_{\text{Vars}} \cdot n_{\text{Ensem}} \cdot n_{\text{Patch1}} \cdot n_{\text{Patch2}}$ , of the fields with edge values set by interpolation.

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

```

118 [~,ny,~,~,~,Ny] = size(patches.y);
119 [nx,~,~,~,Nx,~] = size(patches.x);
120 nEnsem = patches.nEnsem;
121 nVars = round( numel(u)/numel(patches.x) ...
122     /numel(patches.y)/nEnsem );
123 assert(numel(u) == nx*ny*Nx*Ny*nVars*nEnsem ...
124     , 'patchEdgeInt2: input u has wrong size for parameters')
125 u = reshape(u,[nx ny nVars nEnsem Nx Ny]);

```

**Implement multiple width edges by folding** Subsample  $x, y$  coordinates, noting it is only differences that count *and* the microgrid  $x, y$  spacing must be uniform.

```

136 x = patches.x;
137 y = patches.y;
138 if mean(patches.nEdge)>1
139     mx = patches.nEdge(1);
140     my = patches.nEdge(2);
141     x = x(1:mx:nx,:,:,:, :, :);
142     y = y(:,1:my:ny,:,:,:, :, :);
143     nx = nx/mx;
144     ny = ny/my;
145     u = reshape(u,mx,nx,my,ny,nVars,nEnsem,Nx,Ny);
146     nVars = nVars*mx*my;
147     u = reshape( permute(u,[2:2:4 1:2:3 5:8]) ...
148                 ,nx,ny,nVars,nEnsem,Nx,Ny);
149 end%if patches.nEdge

```

**Staggered grid** 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.

```

162 if patches.stag % transform by doubling the number of fields
163 error('staggered grid not yet implemented????')
164 v=nan(size(u)); % currently to restore the shape of u
165 u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
166 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
167 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
168 r=r/2; % ratio effectively halved
169 nPatch=nPatch/2; % halve the number of patches
170 nVars=nVars*2; % double the number of fields
171 else % the values for standard spectral
172     stagShift = 0;
173     iV = 1:nVars;
174 end%if patches.stag

```

#### 4.3.1 Interpolate over the two successive directions

Only use the interior values of the fields for interpolating to the edges.

```
186 u = u(2:nx-1,2:ny-1,:,:,:,:,:);
```

Interpolate in turn, the edge or mid-patch edges normal to the  $x, y$ -directions, in this way we naturally fill-in corner values.

```

194 u = patchEdgeIntCore(1,u,x,patches,stagShift ...
195 ,1,nx,(ny-2)*nVars,nEnsem,1,Nx,Ny,patches.le,patches.ri);
196 u = patchEdgeIntCore(2,u,y,patches,stagShift ...
197 ,nx,ny, nVars,nEnsem,Nx,Ny,1,patches.bo,patches.to);

```

Restore array  $u$  to its original shape.

```
203 u = reshape(u,nx,ny,nVars,nEnsem,Nx,Ny);
```

**Unfold multiple edges** No need to restore  $x, y, z$ .

```

214 if mean(patches.nEdge)>1
215     nVars = nVars/(mx*my);
216     u = reshape( u ,nx,ny,mx,my,nVars,nEnsem,Nx,Ny);
217     nx = nx*mx;
218     ny = ny*my;
219     u = reshape( permute(u,[3 1 4 2 5:8]) ...
220 ,nx,ny,nVars,nEnsem,Nx,Ny);
221 end%if patches.nEdge

```

Fin, returning the 6D array of field values with interpolated edges.

```
229 end% function patchEdgeInt2
```

## 4.4 wave2D: example of a wave on patches in 2D

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  $\mapsto$  patchSys2  $\mapsto$  wavePDE
3. process results

Establish the global data struct `patches` to interface with a function coding the wave PDE: to be solved on  $2\pi$ -periodic domain, with  $9 \times 9$  patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25 (big enough for visualisation), and with a  $5 \times 5$  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 if0urCf2eps([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')
147 [ts,uv] = ode23( @patchSys2,[0 6],uv0(:));
148 else % octave version is slower
149 [ts,uv] = ode0cts(@patchSys2,linspace(0,6),uv0(:));
150 end

```

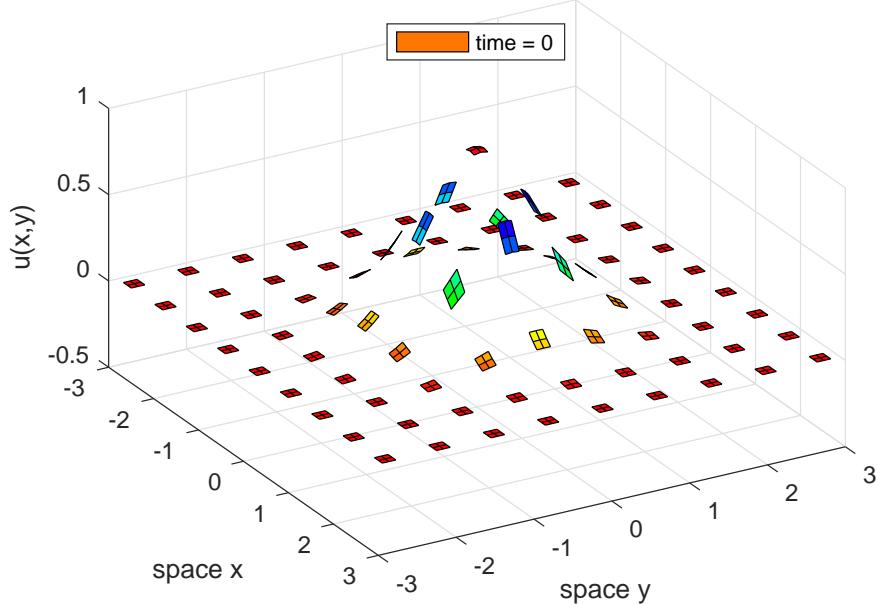
Animate the computed simulation to end with [Figure 4.4](#). Because of the very small time-steps, subsample to plot at most 100 times.

```

158 di = ceil(length(ts)/100);

```

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  $t = 2$ .



```

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(usurf,'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  $\ddot{v}_{ijkl} = \frac{1}{\delta x^2}(u_{i+1,j,k,l} - 2u_{i,j,k,l} + u_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u_{i,j+1,k,l} - 2u_{i,j,k,l} + u_{i,j-1,k,l})$ .

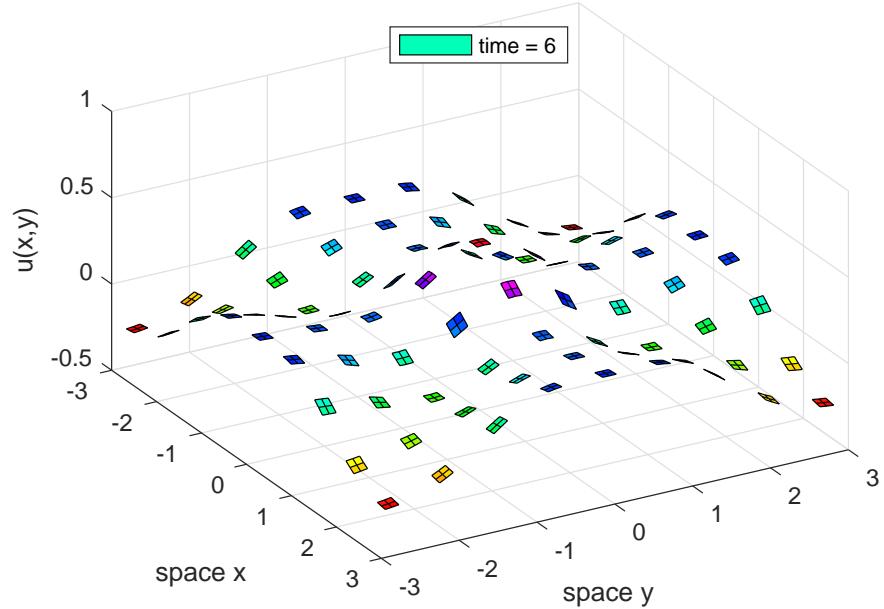
```

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,:);
21   uvt(i,j,2,:) = diff(uv(:,j,1,:),2,1)/dx^2 ...
22     +diff(uv(:,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

```

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.



```
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] = ode0cts(@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  $\epsilon$ . 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  $\epsilon$ .

```
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  $1024\epsilon$ . 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  $8 \times 8$  patches to have  $4 \times 4$  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 HepsiolonRatio = 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.

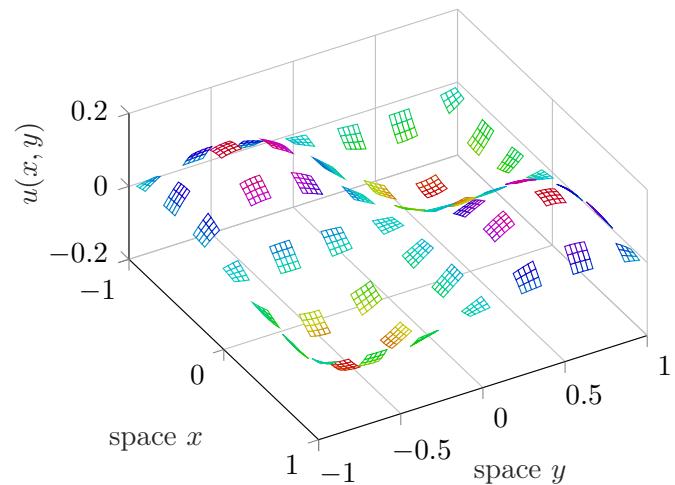
```

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.<sup>8</sup> §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  $7 \times 7$  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.

<sup>8</sup> <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 `NaN`s, 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

```

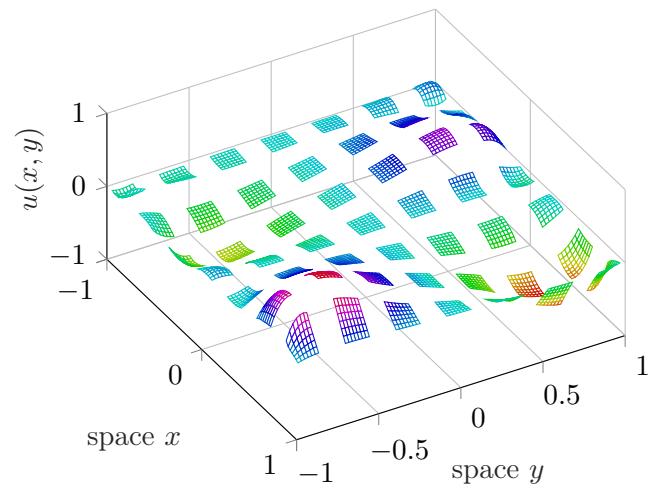
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one  $,:$ .

```

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.<sup>9</sup> §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  $2\epsilon$  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  $\epsilon$ , 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
```

<sup>9</sup> <http://arxiv.org/abs/2211.13731>

**Patch configuration** Say use  $7 \times 7$  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 3.19](#)), 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

```

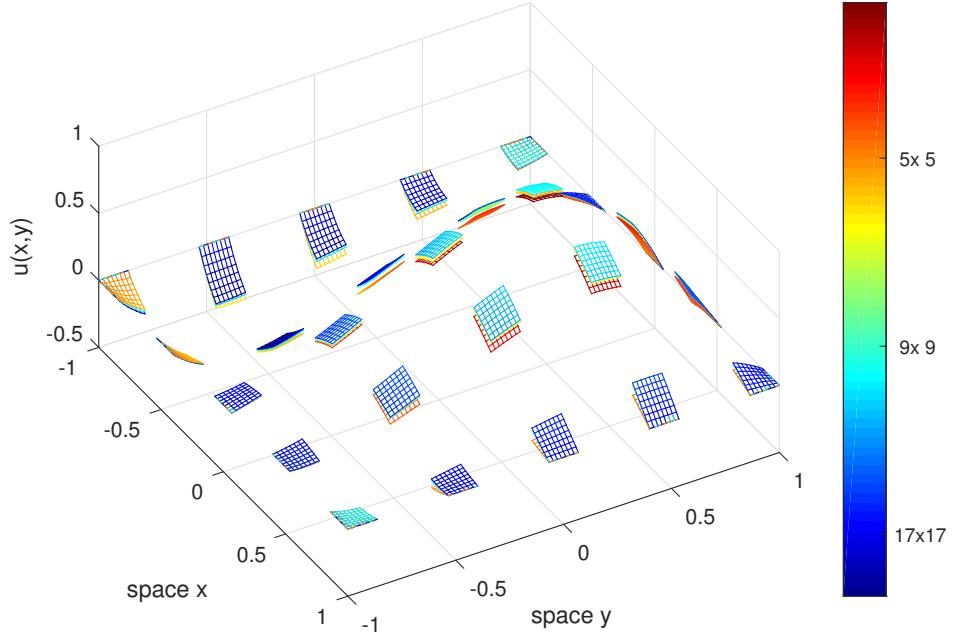
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one  $,:$ .

```

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.<sup>10</sup> §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  $\epsilon$  (here  $\epsilon \approx 0.24, 0.12, 0.06, 0.03$ ), of

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

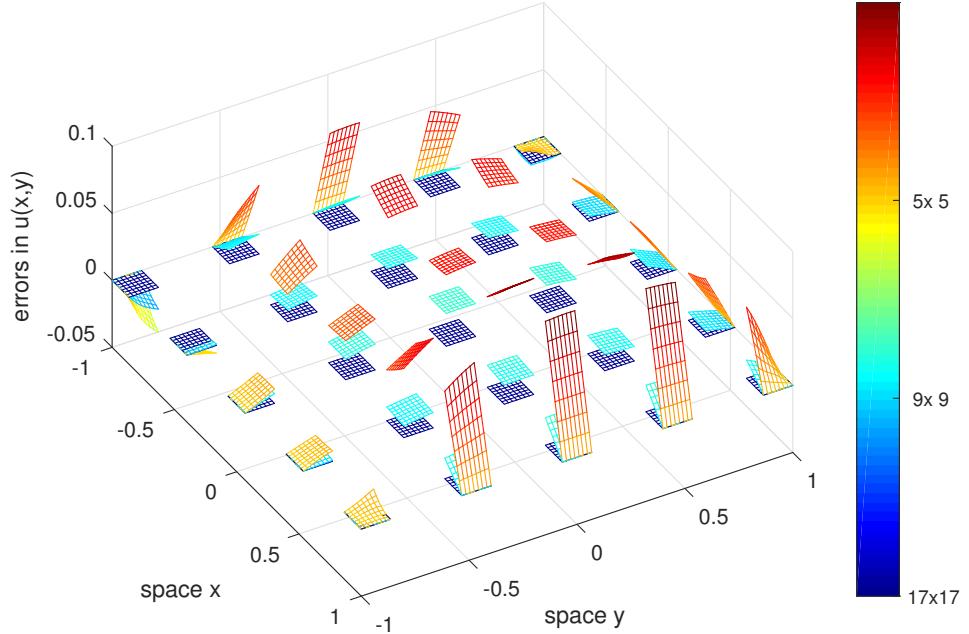
and for forcing  $f := 10(x + y + \cos \pi x)$  (for which the solution has magnitude up to one).<sup>11</sup>

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

<sup>10</sup> <http://arxiv.org/abs/2211.13731>

<sup>11</sup> 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.<sup>12</sup> 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  $\epsilon$ , 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 3.19](#)).

```

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  $\max(\text{abs}())$ .

```

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  $,:$

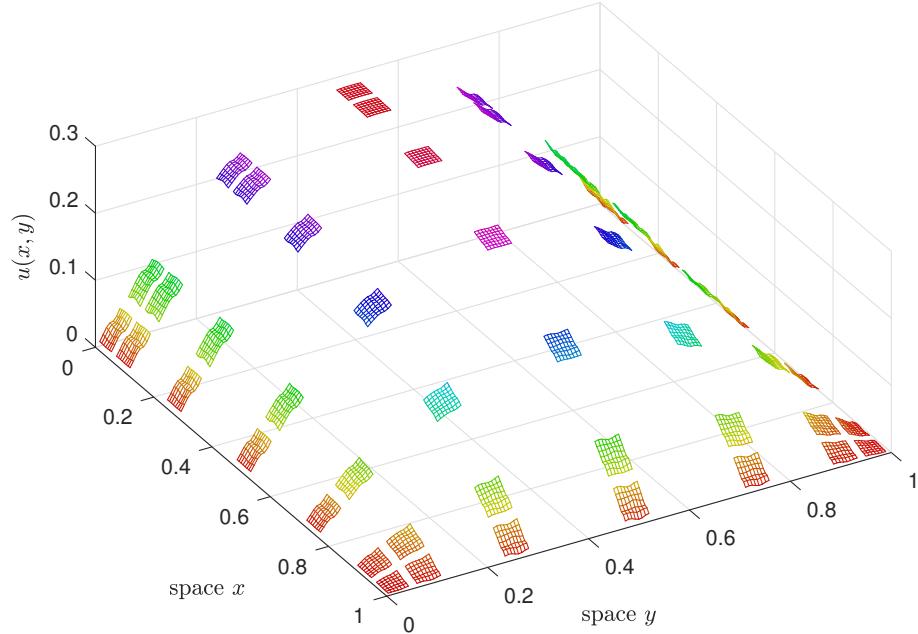
```

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  $\epsilon$  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  $\epsilon$ , 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  $7 \times 7$  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 3.19](#)), 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

```

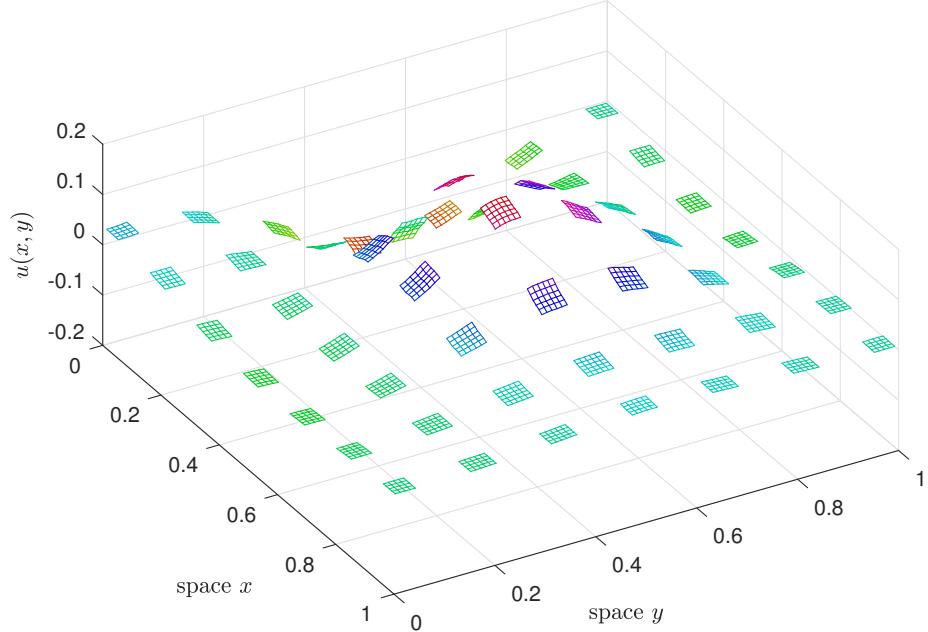
Compute the time derivatives via stored forcing and coefficients. Easier to code by conflating the last four dimensions into the one  $,:$ .

```

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.<sup>13</sup> §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,  $\mu_1(x, y) \in [0.01, 0.1]$ <sup>14</sup> 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
```

<sup>13</sup> <http://arxiv.org/abs/2211.15221>

<sup>14</sup> More interesting microscale structure arises here for  $\mu_1$  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  $\epsilon$ , 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  $7 \times 7$  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 3.19](#)).

```
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  $,:$ .

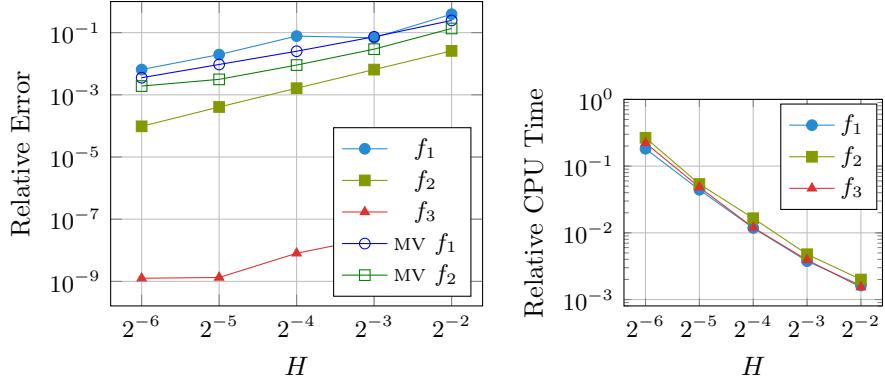
```

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.

### 4.13 SwiftHohenberg2dPattern: patterns of the Swift–Hohenberg PDE in 2D on patches

Figures 4.12 to 4.17 show an example simulation in time generated by the patch scheme applied to the patterns arising from the 2D Swift–Hohenberg PDE.

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

$$\partial_t u = -(1 + \nabla^2/k_0^2)^2 u + Ra u - u^3, \quad (4.1)$$

with various boundary conditions at  $x, y = 0, L$ . For  $Ra$  just above critical, say  $Ra = 0.1$ , the system rapidly evolves to spatial quasi-periodic solutions with period  $\approx 0.24$  when wavenumber parameter  $k_0 = 26$ . These spatial oscillations are here resolved on a micro-grid of spacing 0.042. On medium times these spatial oscillations grow to near equilibrium amplitude of  $\sqrt{Ra}$ , and over very long times the phases of the oscillations evolve in space to adapt to the boundaries.

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

```

42 clear all
43 cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
44 basename = ['r' num2str(floor(1e5*rem(now,1))) mfilename]
45 %global OurCf2eps, OurCf2eps=true %optional to save plots
46 Ra = 0.2 % Ra>0 leads to patterns
47 nGapFac = 2
48 waveLength = 0.5/nGapFac
49 nPtsPeriod = 6
50 dx = waveLength/nPtsPeriod
51 k0 = 2.1*pi/waveLength

```

The above factor 2.1 is close to  $3/\sqrt{2} = 2.1213$  for which  $(\pm 1, \pm 2)$  modes have same linear growth-rate as  $(\pm 2, 0)$  modes.

Establish global data struct `patches` for the Swift–Hohenberg PDE on some square domain. For simplicity, use five patches in each direction. Quartic (fourth-order) interpolation `ordCC = 4` provides values for the inter-patch coupling conditions. Set `bcOffset` for different boundary conditions around the square domain.

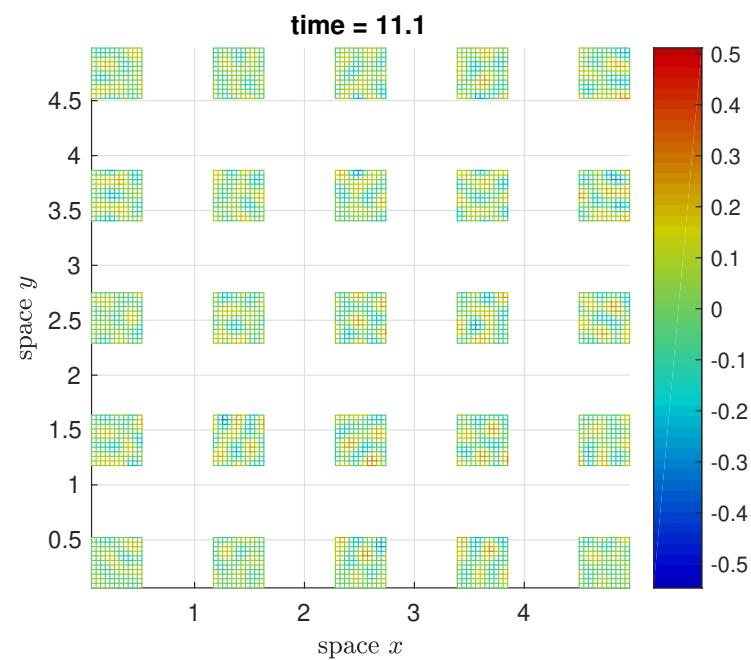
```

67 nPatch = 5
68 nSubP = 2*nPtsPeriod+4
69 Len = nPatch;
70 ordCC = 4;
71 dom.type='equispace';
72 dom.bcOffset=[0.5 0.5;1.0 1.5]
73 patches = configPatches2(@SwiftHohenbergPDE,[0 Len],dom ...
74 ,nPatch,ordCC,dx,nSubP,'EdgyInt',true,'nEdge',2);
75 xs=squeeze(patches.x);
76 ys=squeeze(patches.y);

```

Figure 4.12:

pattern field  $u(x, y, t)$  in the patch scheme applied to a microscale discretisation of the 2D Swift-Hohenberg PDE. At this early time much of the random sub-patch microstructure has decayed leaving some random marginal modes starting to grow.



#### 4.13.0.1 Simulate in time

Set an initial condition, and here integrate forward in time using a standard method for stiff systems. Integrate the interface `patchSys2` (Section 4.2) to the microscale differential equations (despite the extreme stiffness, `ode23` is ten times quicker than `ode15s`). Because pattern evolution is eventually phase-diffusion, here sample the pattern at quadratically varying times.

```

93 fprintf('\n**** Simulate in time\n')
94 u0 = 0.3*( -1+2*rand(size(patches.x+patches.y)) );
95 Ts=400*linspace(0,1,97).^2;
96 tic
97 [ts,us] = ode23(@patchSys2, Ts, u0(:), [], patches,k0,Ra);
98 simulateTime = toc
99 us = reshape(us',nSubP,nSubP,nPatch,nPatch,[]);

```

Plot the simulation such as that shown in Figures 4.12 to 4.17 First, reshape the data, omitting edge values.

```

135 xs([1:2 end-1:end],:) = nan;
136 ys([1:2 end-1:end],:) = nan;
137 us = reshape( permute(us,[1 3 2 4 5]) ...
138 ,nSubP*nPatch,nSubP*nPatch,[]);
139 uRange=[min(us(:)) max(us(:))];

```

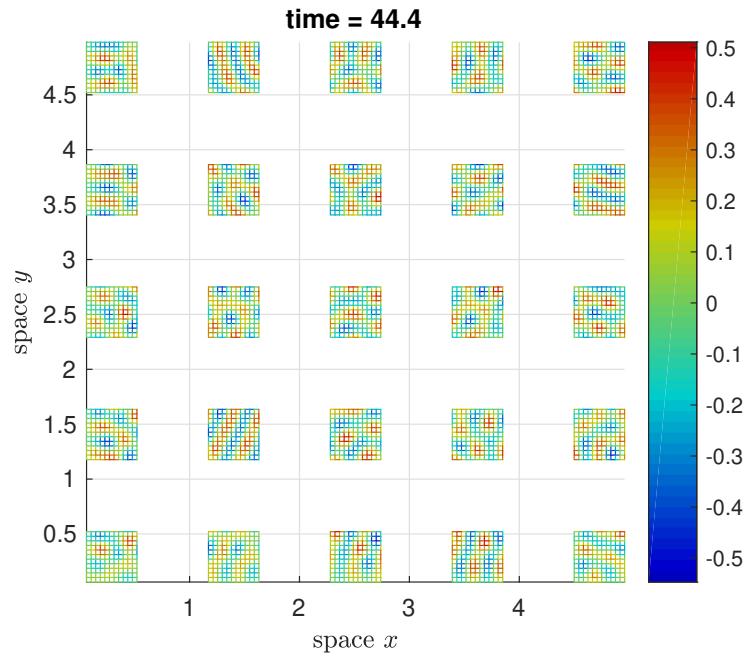
Second, plot six examples of the evolving pattern, equi-spaced in time-index.

```

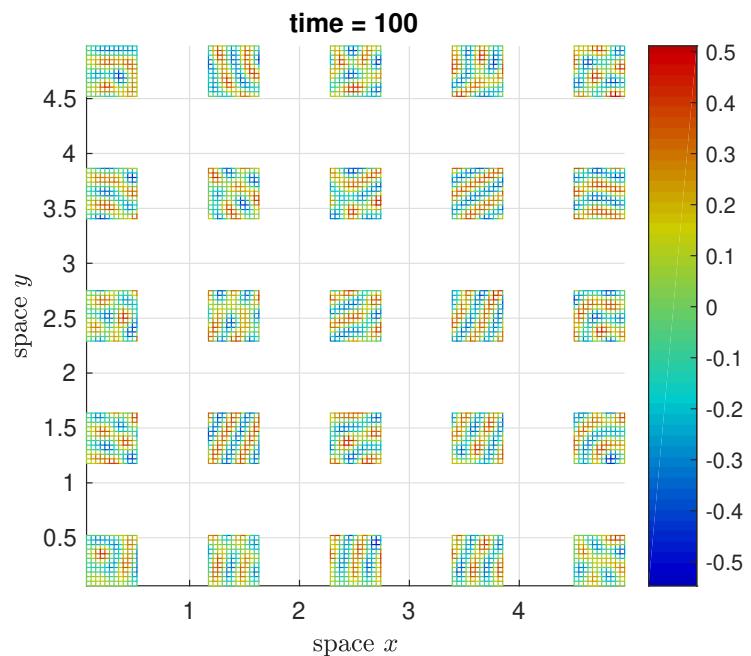
146 plots = round( 1+linspace(0,1,7)*(numel(ts)-1) )
147 for p=2:numel(plots)
148 figure(p),clf
149 mesh(xs(:,ys(:,us(:,:,plots(p))))')

```

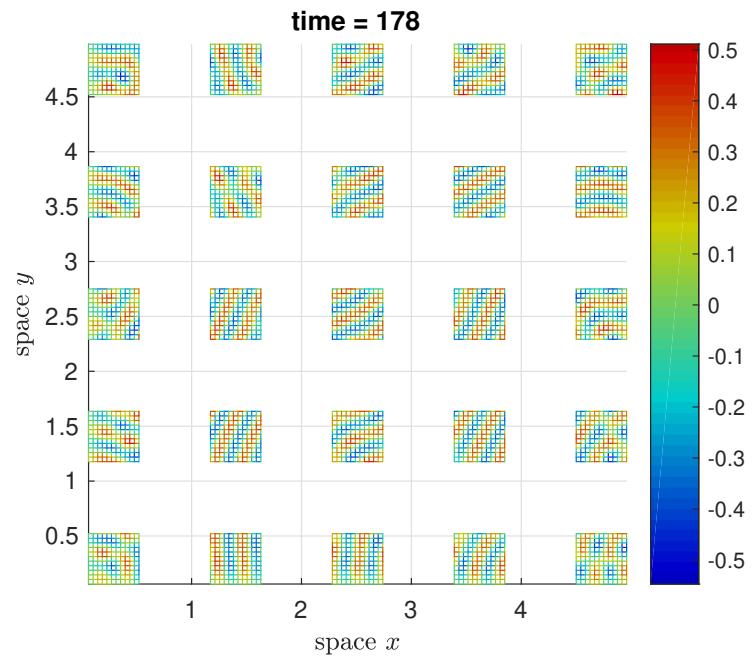
*Figure 4.13:* pattern field  $u(x, y, t)$  in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. By now the local sub-patch patterns have reached a quasi-equilibrium amplitude.



*Figure 4.14:* pattern field  $u(x, y, t)$  in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Patterns within the patches are evolving to the preferred rolls, but with weak coupling to other patches.

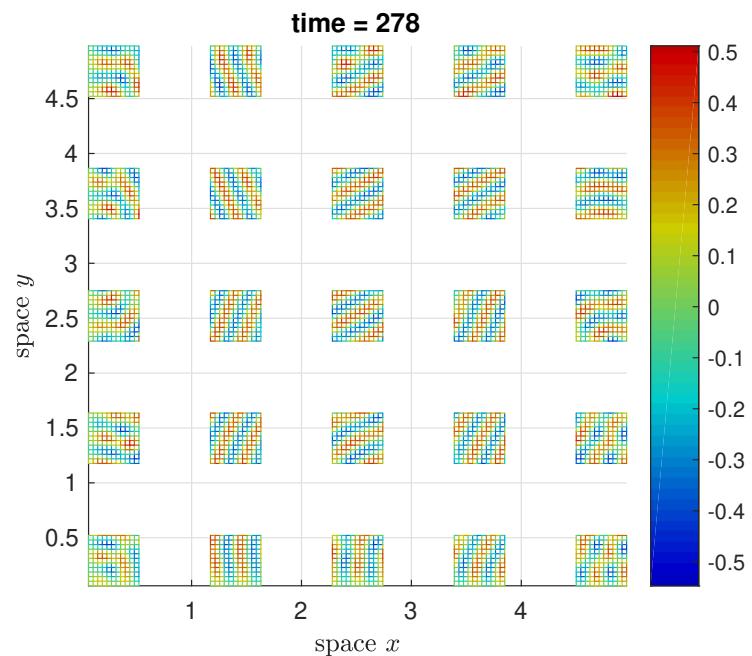


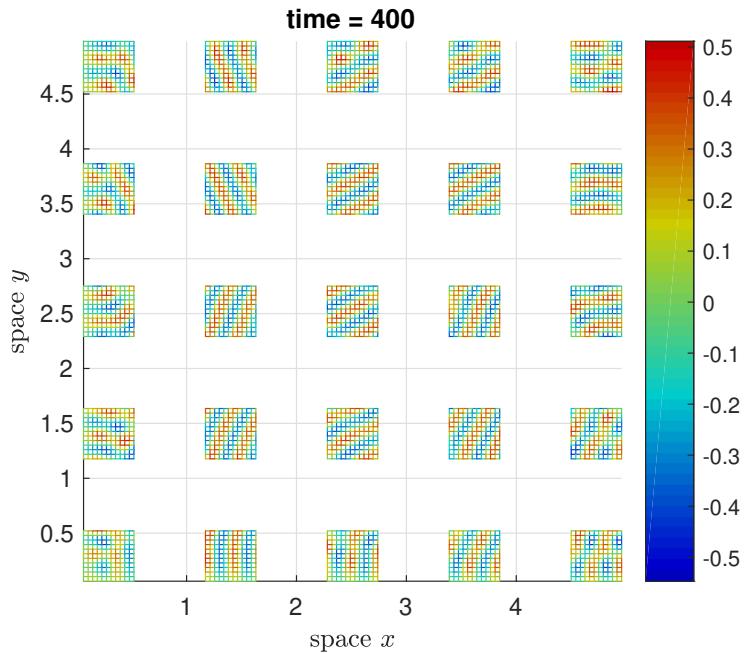
*Figure 4.15:*  
 pattern field  $u(x, y, t)$  in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE. Can see different effects arising at different types of boundaries.



*Figure 4.16:*  
 pattern field  $u(x, y, t)$  in the patch scheme applied to a microscale discretisation of the 2D Swift–Hohenberg PDE.

...





*Figure 4.17:*  
pattern field  
 $u(x, y, t)$  in the  
patch scheme  
applied to a  
microscale dis-  
cretisation of  
the 2D Swift–  
Hohenberg PDE.

...

```

150     axis equal, view(0,90)
151     caxis(uRange), colormap(cMap), colorbar
152     xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y,t)$')
153     title(['time = ' num2str(ts(plots(p)),3)])
154     ifOurCf2eps([basename num2str(p)], [12 11])
155 end%for p

```

Third, plot animation in time: starts after a key press.

```

161 %%
162 figure(1),clf
163 cf=mesh(xs(:,ys(:,us(:,:,1)));
164 axis equal, view(0,90)
165 caxis(uRange), colormap(cMap), colorbar
166 xlabel('space $x$'), ylabel('space $y$'), zlabel('$u(x,y,t)$')
167 title(['time = ' num2str(ts(1),3)])
168 ca=gca;
169 disp('Press any key to start animation'), pause
170 for p=2:numel(ts)
171     cf.ZData=us(:,:,p)';
172     cf.CData=us(:,:,p)';
173     ca.Title.String=['time = ' num2str(ts(p),3)];
174     pause(0.1)
175 end

```

Fin.

#### 4.13.1 The Swift–Hohenberg PDE and BCs inside patches

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

```

189 function ut=SwiftHohenbergPDE(t,u,patches,k0,Ra)
190     dx=diff(patches.x(1:2)); % microscale spacing
191     dy=diff(patches.y(1:2)); % microscale spacing
192     i=3:size(u,1)-2; % interior points in patches
193     j=3:size(u,2)-2; % interior points in patches

```

Code various boundary conditions. For slightly simpler coding, squeeze out the two singleton dimensions.

```

200     u = squeeze(u);
201     u(1:2,:,1,:)=0; % u=u_x=0 at x=0
202     u(:,1:2,:,1)=0; % u=u_y=0 at y=0
203     u(end-1,:,:end,:)=0; % u=0 at x=L
204     u(end ,:,:end,:)=-u(end-2,:,:end,:); % u_x=0 at x=L
205     u(:,end-1,:,:end)=-u(:,end-2,:,:end); % u_y=0 at y=L
206     u(:,end ,:,:end)=-u(:,end-3,:,:end); % u_yyy=0 at y=L

```

Here code straightforward centred discretisation in space.

```

212     ut=nan+u; % preallocate output array
213     v = u(2:end-1,2:end-1,:,:,:) ...
214         +( diff(u(:,2:end-1,:,:,:),2,1)/dx^2 ...
215             +diff(u(2:end-1,:,:,:,:),2,2)/dy^2 )/k0^2;
216     ut(i,j,:,:,:) = -( v(2:end-1,2:end-1,:,:,:) ...
217         +( diff(v(:,2:end-1,:,:,:),2,1)/dx^2 ...
218             +diff(v(2:end-1,:,:,:,:),2,2)/dy^2 )/k0^2 ) ...
219         +Ra*u(i,j,:,:,:) -u(i,j,:,:,:).^3;
220 end

```

#### 4.13.2 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 = 20

```

##### 4.13.2.1 Check divided difference interpolation

```

36 fprintf('\n\n**** Check divided difference interpolation\n')
37 pause(1)

```

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.)

```

46 maxErrors=[];
47 for realisation = 1:nRealise
48     nEdge = randi(3,1,2)% =1,2, or 3
49     edgyInt = (rand>0.5)
50     Lx = 1+3*rand; Ly = 1+3*rand;
51     xyLim = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
52     nSubP = nEdge.*((2-edgyInt)*randi(3,1,2)+1+edgyInt )
53     ordCC = 2*randi(4)
54     nPatch = ordCC+randi(4,1,2)
55     dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
56     configPatches2(@sin,xyLim,'equispace',nPatch,ordCC ...
57                 ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

Second, displace patches to a random non-uniform spacing.

```

63 Hx = diff(patches.x(1,1,:,:,:,1:2,1));
64 patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,nPatch(1),1)-0.5);
65 Hx = squeeze( diff(patches.x(1,1,:,:,:,1)) );% for information only
66 Hy = diff(patches.y(1,1,:,:,:,1:2));
67 patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,nPatch(2))-0.5);
68 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.

```

81 ox=min(ordCC,nPatch(1)-1);
82 oy=min(ordCC,nPatch(2)-1);
83 [ps,qs]=ndgrid(0:ox,0:oy);
84 ps=reshape(ps,1,1,[ ]);
85 qs=reshape(qs,1,1,[ ]);
86 cs=2*rand(size(ps))-1;
87 u0=cs.*patches.x.^ps.*patches.y.^qs;

```

Then evaluate the interpolation, setting edges to `inf` for error checking.

```

94 u=u0;
95 u([1:nEdge(1) end-nEdge(1)+1:end],:,:) = inf;
96 u(:,[1:nEdge(2) end-nEdge(2)+1:end],:) = inf;
97 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 `Nan`s here. High-order interpolation seems to be more affected by round-off so relax error size.

```

107 error = ui-u0;
108 hist(log10(abs(error(abs(error)>1e-20))),-20:-7)
109 xlabel('log10 error'), pause(0.3)%??
110 maxError=max(abs(error(:)))
111 maxErrors=[maxErrors maxError];

```

```

112     assert(maxError<3e-12*4^ordCC ...
113     , 'failed divided difference interpolation')
114     disp('*** This divided difference test passed')

```

End the for-loops over various parameters.

```

121 end% for realisation
122 maxMaxErrorDividedDiffs = max(maxErrors)
123 disp('***** Passed all divided difference interpolation')
124 pause(1)

```

#### 4.13.2.2 Test standard spectral interpolation

```

138 fprintf('\n\n***** Test standard spectral interpolation\n')
139 pause(1)

```

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
146 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.

```

154 nEdge=randi(3,1,2)% =1,2, or 3
155 edgyInt = (rand>0.5)
156 Lx = 1+3*rand, Ly = 1+3*rand
157 xyLim = [0 Lx 0 Ly]-[rand*[1 1] rand*[1 1]]
158 nSubP = nEdge.*((2-edgyInt)*randi(3,1,2)+1+edgyInt )
159 nPatch = randi([3 6],1,2)
160 dx = [Lx Ly]./nPatch./nSubP.*rand(1,2)/2
161 configPatches2(@sin,xyLim,'periodic',nPatch,0 ...
162     ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

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.

```

172 nV=randi(3)
173 [nx,Nx]=size(squeeze(patches.x));
174 [ny,Ny]=size(squeeze(patches.y));
175 u0=nan(nx,ny,nV,1,Nx,Ny);
176 for iV=1:nV
177     kx=randi([0 floor((nPatch(1)-1)/2)])
178     ky=randi([0 floor((nPatch(2)-1)/2)])
179     phix=pi*rand*(2*kx~=nPatch(1))
180     phiy=pi*rand*(2*ky~=nPatch(2))
181     % generate 6D array via auto-replication
182     u0(:,:,:iV,1,:,:)=sin(2*pi*kx*patches.x/Lx+phix) ...
183         .*sin(2*pi*ky*patches.y/Ly+phiy);
184 end

```

Copy and `nan` the edges, then interpolate

```
190 u=u0;
191 u([1:nEdge(1) end-nEdge(1)+1:end],:,:)=nan;
192 u(:,[1:nEdge(2) end-nEdge(2)+1:end],:,:)=nan;
193 u=patchEdgeInt2(u(:));
```

Compute difference. If there is an error in the interpolation, then abort the script for checking: please record parameter values and inform us.

```
201 error = u-u0;
202 assert(all(~isnan(error(:))),'found nans in the error!')
203 hist(log10(abs(error(abs(error)>1e-20))),-20:-7)
204 xlabel('log10 error'), pause(0.3)%??
205 normError=norm(error(:))
206 assert(normError<1e-12, '2D spectral interpolation failed')
207 disp('*** This spectral test passed')

End the for-loop over realisations

214 end
215 disp('***** All the spectral tests passed')
216 pause(1)
```

#### 4.13.2.3 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.)

```
234 for realisations=1:nRealise
235     nEdge = randi(3,1,2)% =1,2, or 3
236     edgyInt = (rand>0.5)
237     nSubP = nEdge.*((2-edgyInt)*randi(3,1,2)+1+edgyInt )
238     ordCC = 2*randi(4)
239     nPatch = ordCC+randi(4,1,2)
240     xyLim=5*[-rand(1,2); rand(1,2)]
241     dx = diff(xyLim)./nPatch./nSubP.*rand(1,2)/2
242     configPatches2(@sin,xyLim,'periodic',nPatches,ordCC ...
243                 ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);
```

**Check multiple fields simultaneously** Set profiles to be various powers of  $x$ ,  $ps$ , and store as different ‘variables’ at each point.

```
252 [ps,qs]=meshgrid(0:ordCC);
253 ps=reshape(ps,1,1,[ ]); qs=reshape(qs,1,1,[ ]);
254 cs=2*rand(size(ps))-1;
255 u0=cs.*patches.x.^ps.*patches.y.^qs;
```

Then evaluate the interpolation.

```
261 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.

```
269 I=ordCC/2+1:nPatch(1)-ordCC/2;
270 J=ordCC/2+1:nPatch(2)-ordCC/2;
271 error=ui(:,:, :, I, J)-u0(:,:, :, I, J);
272 assert(all(~isnan(error(:))), 'found nans in the error!')
273 hist(log10(abs(error(abs(error)>1e-20))), -20:-7)
274 xlabel('log10 error'), pause(0.3)%%
275 normError=norm(error(:))
276 assert(normError<5e-9 ...
277 , 'failed finite stencil interpolation')
278 disp('*** This finite stencil test passed')

End the for-loops over various parameters.

285 end %for realisations
286 disp('***** Passed all standard polynomial interpolation')
```

#### 4.13.2.4 Finished

If no error messages, then all OK.

```
300 disp('***** All the interpolation tests successful')
```

---

## 5 Patches in 3D space

---

## 5.1 configPatches3(): configures spatial patches in 3D

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys3()`, and possibly other patch functions. [Sections 5.1.1](#) and [5.4](#) list examples of its use.

```
20 function patches = configPatches3(fun,Xlim,Dom ...
21     ,nPatch,ordCC,dx,nSubP,varargin)
22 version = "2023-10-18";
```

**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.

But, if `nPatch(n) = 1`, then in the spatial direction `n` the domain is the mean of `Xlim(2*n-1:2*n)` plus/minus half the patch-width in that direction.

- `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 array, 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`, char-array, of either `'periodic'` (the default, equivalently `'periodic'`), `'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 leftmost/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` determines the number of spatial patches: if scalar, then use the same number of patches in all three directions, otherwise `nPatch(1:3)` gives the number ( $\geq 1$ ) 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—adjusted a little when `nEdge` > 1. So either if not `EdgyInt` then `ratio = 1/2` means the patches abut and `ratio = 1` is overlapping patches as in holistic discretisation, or if `EdgyInt` then `ratio = 1` means the patches abut. Small `ratio` should greatly reduce computational time.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in all three directions, otherwise `nSubP(1:3)` gives the number in each direction. If not using `EdgyInt`, then `nSubP ./nEdge` must be odd integer(s) so that there is/are centre-patch lattice planes. So for the defaults of `nEdge = 1` and not `EdgyInt`, then `nSubP` must be odd.
- ‘`nEdge`’, *optional* (integer—scalar or three element), default=1, the width of face values set by interpolation at the face regions of each patch. If two elements, then respectively the width in  $x$ ,  $y$ -directions.

The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- **'EdgyInt'**, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom/front/back face-values from right/left/bottom/top/back/front next-to-face values. If false or omitted, then interpolate from centre-patch planes.
- **'nEnsem'**, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- **'hetCoeffs'**, *optional*, default empty. Supply a 3D or 4D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size  $m_x \times m_y \times m_z \times n_c$ , where  $n_c$  is the number of different arrays of coefficients. For example, in heterogeneous diffusion,  $n_c = 3$  for the diffusivities in the *three* different spatial directions (or  $n_c = 6$  for the diffusivity tensor). The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the  $n_c$  coefficients being  $n_c$  parameters in some macroscale formula.
  - If `nEnsem` = 1, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the (1, 1, 1)-point in each patch. Best accuracy usually obtained when the periodicity of the coefficients is a factor of `nSubP-2*nEdge` for `EdgyInt`, or a factor of `(nSubP-nEdge)/2` for not `EdgyInt`.
  - If `nEnsem` > 1 (value immaterial), then reset `nEnsem` :=  $m_x \cdot m_y \cdot m_z$  and construct an ensemble of all  $m_x \cdot m_y \cdot m_z$  phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in  $x, y, z$ -directions, respectively, then this coupling cunningly preserves symmetry.
- **'parallel'**, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

If true, and it requires that you have MATLAB's Parallel Computing Toolbox, then `configPatches3` will distribute the patches over multiple CPUS/cores. In MATLAB, only one array dimension can be split in the distribution, so `configPatches3` 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.<sup>1</sup>

```
233 if nargout==0, global patches, end
234 patches.version = version;
```

- .fun is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` or `fun(t,u,patches,...)` that computes the time derivatives (or steps) on the patchy lattice.
- .ordCC is the specified order of inter-patch coupling.
- .periodic: either true, for interpolation on the macro-periodic domain; or false, for general interpolation by divided differences over non-periodic domain or unevenly distributed patches.
- .stag is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- .Cwtsr and .Cwtsl are the `ordCC` × 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 × `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, for macro-periodic conditions, are the size ratios of every patch.
- .nEdge 1 × 3, is the width of face values set by interpolation at the face regions of each patch, in the  $x, y, z$ -directions respectively.
- .le, .ri, .bo, .to, .ba, .fr determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- .cs either
  - [] 0D, or
  - if `nEnsem` = 1,  $(nSubP(1)-1) \times (nSubP(2)-1) \times (nSubP(3)-1) \times n_c$  4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1,  $(nSubP(1)-1) \times (nSubP(2)-1) \times (nSubP(3)-1) \times n_c \times m_x m_y m_z$  5D array of  $m_x m_y m_z$  ensemble of phase-shifts of the microscale heterogeneous coefficients.

---

<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.

- `.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

```
323 if nargin==0
324 disp('With no arguments, simulate example of heterogeneous wave')
```

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

1. configPatches3
2. ode23 integrator  $\mapsto$  patchSys3  $\mapsto$  user's PDE
3. process results

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the macro-wave time scale is roughly one.

```
342 mPeriod = [2 2 2];
343 cHetr = exp(0.9*randn([mPeriod 3]));
344 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.

```
356 global patches
357 patches = configPatches3(@heteroWave3, [-pi pi] ...
358 , 'periodic', 5, 0, 0.22, mPeriod+2, 'EdgyInt', true ...
359 , 'hetCoeffs', cHetr);
```

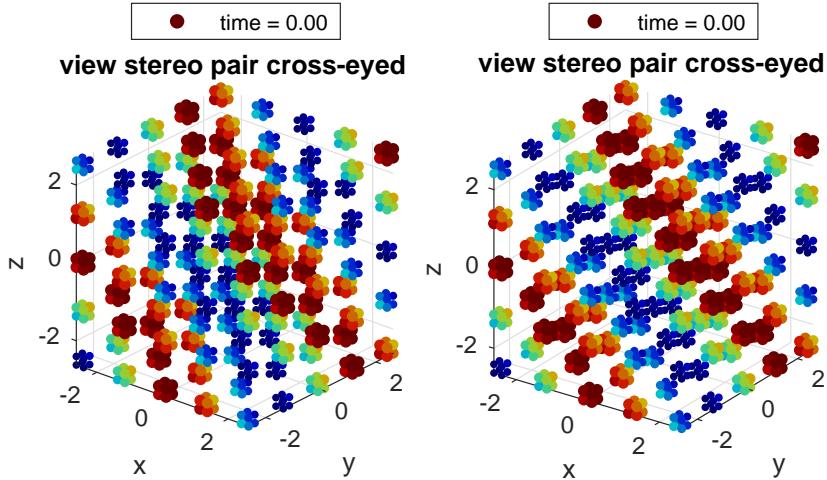
Set a wave initial state using auto-replication of the spatial grid, and as Figure 5.1 shows. This wave propagates diagonally across space. Concatenate the two  $u, v$ -fields to be the two components of the fourth dimension.

```
369 u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
370 v0 = -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
371 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).

```
388 disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
389 if ~exist('OCTAVE_VERSION', 'builtin')
390 [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
391 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  $t = 6$ .



```

392     lsode_options('absolute tolerance',1e-4);
393     lsode_options('relative tolerance',1e-4);
394     [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
395 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.

```

409 figure(1), clf, colormap(0.8*jet)
410 xs = patches.x+0*patches.y+0*patches.z;
411 ys = patches.y+0*patches.x+0*patches.z;
412 zs = patches.z+0*patches.y+0*patches.x;
413 if 1, xs([1 end],:,:)=nan;
414     xs(:,[1 end],:,:)=nan;
415     xs(:,:,1 end,:)=nan;
416 end;%option
417 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.

```

425 pix = @(u) 15*abs(u)+7;
426 col = @(u) sign(u).*abs(u);

```

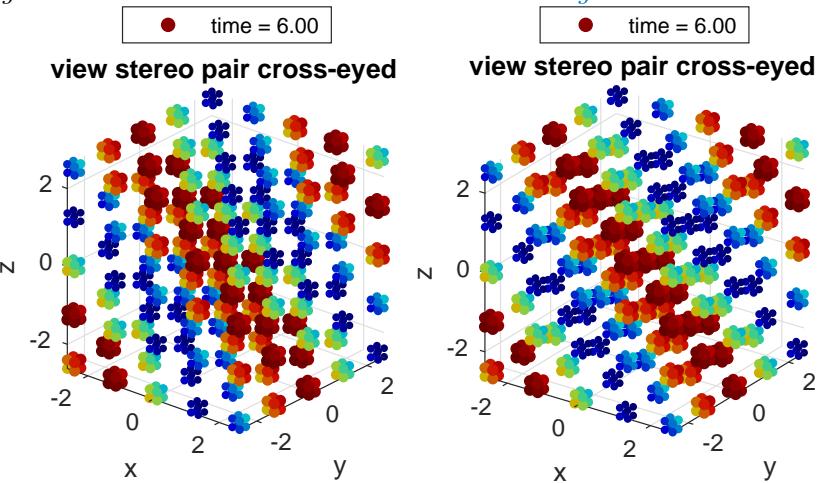
Loop to plot at each and every time step.

```

432 for i = 1:length(ts)
433     uv = patchEdgeInt3(us(i,:));
434     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.



```

435 for p=1:2
436 subplot(1,2,p)
437 if (i==1)| exist('OCTAVE_VERSION', 'builtin')
438 scat(p) = scatter3(xs(j),ys(j),zs(j), 'filled');
439 axis equal, caxis(col([0 1])), view(45-5*p,25)
440 xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
441 title('view stereo pair cross-eyed')
442 end % in matlab just update values
443 set(scat(p), 'CData', col(u(j)) ...
444 , 'SizeData', pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
445 legend(['time = ' num2str(ts(i), '%4.2f')], 'Location', 'north')
446 end

```

Optionally save the initial condition to graphic file for Figure 4.1, and optionally save the last plot.

```

454 if i==1,
455 ifOurCf2eps([mfilename 'ic'])
456 disp('Type space character to animate simulation')
457 pause
458 else pause(0.05)
459 end
460 end% i-loop over all times
461 ifOurCf2eps([mfilename 'fin'])

```

Upon finishing execution of the example, exit this function.

```

476 return
477 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

```
494 p = inputParser;
495 fnValidation = @(f) isa(f, 'function_handle'); %test for fn name
496 addRequired(p,'fun',fnValidation);
497 addRequired(p,'Xlim',@isnumeric);
498 %addRequired(p,'Dom'); % too flexible
499 addRequired(p,'nPatch',@isnumeric);
500 addRequired(p,'ordCC',@isnumeric);
501 addRequired(p,'dx',@isnumeric);
502 addRequired(p,'nSubP',@isnumeric);
503 addParameter(p,'nEdge',1,@isnumeric);
504 addParameter(p,'EdgyInt',false,@islogical);
505 addParameter(p,'nEnsem',1,@isnumeric);
506 addParameter(p,'hetCoeffs',[],@isnumeric);
```

```

507 addParameter(p,'parallel',false,@islogical);
508 %addParameter(p,'nCore',1,@isnumeric); % not yet implemented
509 parse(p,fun,Xlim,nPatch,ordCC,dx,nSubP,varargin{:});

```

Set the optional parameters.

```

515 patches.nEdge = p.Results.nEdge;
516 if numel(patches.nEdge)==1
517     patches.nEdge = repmat(patches.nEdge,1,3);
518 end
519 patches.EdgyInt = p.Results.EdgyInt;
520 patches.nEnsem = p.Results.nEnsem;
521 cs = p.Results.hetCoeffs;
522 patches.parallel = p.Results.parallel;
523 %patches.nCore = p.Results.nCore;

```

Initially duplicate parameters for three space dimensions as needed.

```

531 if numel(Xlim)==2,    Xlim = repmat(Xlim,1,3); end
532 if numel(nPatch)==1,  nPatch = repmat(nPatch,1,3); end
533 if numel(dx)==1,      dx = repmat(dx,1,3); end
534 if numel(nSubP)==1,   nSubP = repmat(nSubP,1,3); end

```

Check parameters.

```

541 assert(all(nPatch==round(nPatch))&all(nPatch>0) ...
542     , 'nPatch must be integer(s) and at least one')
543 assert(Xlim(1)<Xlim(2) ...
544     , 'first pair of Xlim must be ordered increasing')
545 assert(Xlim(3)<Xlim(4) ...
546     , 'second pair of Xlim must be ordered increasing')
547 assert(Xlim(5)<Xlim(6) ...
548     , 'third pair of Xlim must be ordered increasing')
549 assert((mod(ordCC,2)==0)|all(patches.nEdge==1) ...
550     , 'Cannot yet have nEdge>1 and staggered patch grids')
551 assert(all(3*patches.nEdge<=nSubP) ...
552     , 'too many edge values requested')
553 assert(all(rem(nSubP,patches.nEdge)==0) ...
554     , 'nSubP must be integer multiple of nEdge')
555 if ~patches.EdgyInt, assert(all(rem(nSubP./patches.nEdge,2)==1) ...
556     , 'for non-edgyInt, nSubP./nEdge must be odd integer')
557 end
558 if (patches.nEnsem>1)&all(patches.nEdge>1)
559     warning('not yet tested when both nEnsem and nEdge non-one')
560 end
561 %if patches.nCore>1
562 %    warning('nCore>1 not yet tested in this version')
563 %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.

```

574 if ~isstruct(Dom), pre2023=any(isnan(Dom));
575 else pre2023=false; end
576 if pre2023, ratio=dx; dx=nan; end

Default macroscale conditions are periodic with evenly spaced patches.

585 if isempty(Dom), Dom=struct('type','periodic'); end
586 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.

594 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.

607 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 bound-
aries if no bcOffset specified when needed. Do so for all three directions
independently.

615 patches.periodic = false(3,1);
616 for p=1:3
617 switch Dom.type(p,:)
618 case {'periodic','periodic'}
619     patches.periodic(p) = true;
620     msg=' not available for Dom.type = periodic';
621     if isfield(Dom,'bcOffset')&&~all(isnan(Dom.bcOffset(:,p)))
622         warning(['bcOffset' msg]), end
623     if isfield(Dom,'X'), warning(['X' msg]), end
624     if isfield(Dom,'Y'), warning(['Y' msg]), end
625     if isfield(Dom,'Z'), warning(['Z' msg]), end
626 case {'equispace','chebyshev'}
627     if ~isfield(Dom,'bcOffset')
628         Dom.bcOffset=nan(2,3); Dom.bcOffset(:,p)=0;
629     else Dom.bcOffset( isnan(Dom.bcOffset(:,p)), ,p)=0;
630     end
631 % for mixed with usergiven, following should still work
632     if numel(Dom.bcOffset)==1
633         Dom.bcOffset=repmat(Dom.bcOffset,2,3); end
634     if numel(Dom.bcOffset)==3
635         Dom.bcOffset=repmat(Dom.bcOffset(:,2,1)); end
636     msg=' not available for Dom.type = equispace or chebyshev';
637     if (p==1)& isfield(Dom,'X'), warning(['X' msg]), end
638     if (p==2)& isfield(Dom,'Y'), warning(['Y' msg]), end
639     if (p==3)& isfield(Dom,'Z'), warning(['Z' msg]), end
640 case 'usergiven'

```

```

641 %     if isfield(Dom,'bcOffset')
642 %     warning('bcOffset not available for usergiven Dom.type'), end
643     msg=' required for Dom.type = usergiven';
644     if p==1, assert(isfield(Dom,'X'),['X' msg]), end
645     if p==2, assert(isfield(Dom,'Y'),['Y' msg]), end
646     if p==3, assert(isfield(Dom,'Z'),['Z' msg]), end
647 otherwise
648     error([Dom.type(p,:) ' is unknown Dom.type'])
649 end%switch Dom.type
650 end%for p

```

#### 5.1.4 The code to make patches

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

```
664 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.

```

673 assert((ordCC>=-1) & (floor(ordCC)==ordCC), ...
674     'ordCC out of allowed range integer>=-1')

```

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

```

681 patches.stag = mod(ordCC,2);
682 assert(patches.stag==0,'staggered not yet implemented??')
683 ordCC = ordCC+patches.stag;
684 patches.ordCC = ordCC;

```

Check for staggered grid and periodic case.

```

690 if patches.stag, assert(all(mod(nPatch,2)==0), ...
691     'Require an even number of patches for staggered grid')
692 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.

```

707 patches.ratio = nan(1,3);
708 for q=1:3
709 qq=2*q-1; qstr=num2str(q);

```

Distribution depends upon `Dom.type`:

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

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

```

723 case {'periodic','periodic'}
724 Q=linspace(Xlim(qq),Xlim(qq+1),nPatch(q)+1);
725 DQ=Q(2)-Q(1);
726 Q=Q(1:nPatch(q))+diff(Q)/2;
727 pEI=patches.EdgyInt;% abbreviation
728 pnE=patches.nEdge(q);% abbreviation
729 if pre2023, dx(q) = ratio(q)*DQ/(nSubP(q)-pnE*(1+pEI))*(2-pEI);
730 else ratio(q) = dx(q)/DQ*(nSubP(q)-pnE*(1+pEI))/(2-pEI);
731 end
732 patches.ratio(q)=ratio(q);

```

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

```

741 case 'equispace'
742 halfWidth=dx(q)*(nSubP(q)-1)/2;
743 if nPatch(q)>1
744 Q=linspace(Xlim(qq)+halfWidth-Dom.bcOffset(qq)*dx(q) ...
745 ,Xlim(qq+1)-halfWidth+Dom.bcOffset(qq+1)*dx(q) ...
746 ,nPatch(q));
747 DQ=diff(Q(1:2));
748 else% nPatch(q)==1
749 warning(['nPatch(' qstr ')==1 so placing one patch at the mean of Xlim']);
750 Q=mean(Xlim(qq:qq+1));
751 DQ=Inf;
752 end%if nPatch(q)
753 width=(1+patches.EdgyInt)/2*(nSubP(q)-1-patches.EdgyInt)*dx;
754 if DQ<width*0.99999
755 warning(['excess equispace patches (double overlap) in dirn ' qstr])
756 end%if

```

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’.<sup>2</sup>

```

773 case 'chebyshev'
774 halfWidth=dx(q)*(nSubP(q)-1)/2;
775 if nPatch(q)>1
776 Q1 = Xlim(1)+halfWidth-Dom.bcOffset(qq)*dx(q);
777 Q2 = Xlim(2)-halfWidth+Dom.bcOffset(qq+1)*dx(q);
778 % 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`.

---

<sup>2</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 integration to a slow manifold (e.g., [Gear et al. 2005a](#), [Gear & Kevrekidis 2005](#), [Maclean et al. 2020](#)). Here maybe the overlapping patches allow for a ‘christmas tree’ approach to the boundary layers. Needs to be explored??

```

787     pEI=patches.EdgeInt; % abbreviation
788     pnE=patches.nEdge(q);% abbreviation
789     width=(1+pEI)/2*(nSubP(q)-pnE*(1+pEI))*dx(q);
790     for b=0:2:nPatch(q)-2
791         DQmin=(Q2-Q1-b*width)/2*( 1-cos(pi/(nPatch(q)-b-1)) );
792         if DQmin>width, break, end
793     end%for b
794     if DQmin<width*0.999999
795         warning(['excess Chebyshev patches (mid-domain overlap) in dirn ', qstr])
796     end%if

```

Assign the centre-patch coordinates.

```

802     Q =[ Q1+(0:b/2-1)*width ...
803           (Q1+Q2)/2-(Q2-Q1-b*width)/2*cos(linspace(0,pi,nPatch(q)-b)) ...
804           Q2+(1-b/2:0)*width ];
805     else% nPatch(q)==1
806         warning(['nPatch(' qstr ')==1 so placing one patch at the mean of Xlim'])
807         Q=mean(Xlim(qq:qq+1));
808     end%if nPatch(q)

```

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

```

817 case 'usergiven'
818     if q==1, Q = reshape(Dom.X,1,[]); end
819     if q==2, Q = reshape(Dom.Y,1,[]); end
820     if q==3, Q = reshape(Dom.Z,1,[]); end
821 end%switch Dom.type

```

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

```

828 if q==1, X=Q; end
829 if q==2, Y=Q; end
830 if q==3, Z=Q; end
831 end%for q

```

**Construct the micro-grids** Fourth, construct the microscale grid in each patch, centred about the given mid-points  $X, Y, Z$ . Reshape the grid to be 8D to suit dimensions (micro,Vars,Ens,macro).

```

847 xs = dx(1)*( (1:nSubP(1))-mean(1:nSubP(1)) );
848 patches.x = reshape( xs'+X ...
849                         ,nSubP(1),1,1,1,1,nPatch(1),1,1);
850 ys = dx(2)*( (1:nSubP(2))-mean(1:nSubP(2)) );
851 patches.y = reshape( ys'+Y ...
852                         ,1,nSubP(2),1,1,1,1,nPatch(2),1);
853 zs = dx(3)*( (1:nSubP(3))-mean(1:nSubP(3)) );
854 patches.z = reshape( zs'+Z ...
855                         ,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.<sup>3</sup> <sup>4</sup>

```

868 if any(patches.periodic)
869 % ratio = reshape(ratio,1,3); % force to be row vector
870 % patches.ratio = ratio;
871 if ordCC>0
872     [Cwtsr,Cwtsl] = patchCwts(patches.ratio,ordCC,patches.stag);
873     patches.Cwtsr = Cwtsr; patches.Cwtsl = Cwtsl;
874 end%if
875 end%if any 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()`.

```

904 nE = patches.nEnsem;
905 patches.le = 1:nE; patches.ri = 1:nE;
906 patches.bo = 1:nE; patches.to = 1:nE;
907 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.

```

919 if ~isempty(cs)
920     [mx,my,mz,nc] = size(cs);
921     nx = nSubP(1); ny = nSubP(2); nz = nSubP(3);
922     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.

```

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

```

<sup>3</sup> **ToDo:** Might sometime extend to coupling via derivative values.

<sup>4</sup> **ToDo:** Is this ratio stuff OK????

But for  $n_{\text{Ensem}} > 1$  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.

```

940     patches.nEnsem = mx*my*mz;
941     patches.cs = nan(nx-1,ny-1,nz-1,nc, mx, my, mz);
942     for k = 1:mz
943         ks = (k:k+nz-2);
944         for j = 1:my
945             js = (j:j+ny-2);
946             for i = 1:mx
947                 is = (i:i+nx-2);
948                 patches.cs(:,:, :, :, i, j, k) = cs(is,js,ks,:);
949             end
950         end
951     end
952     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.

```

962     mmx=(0:mx-1)'; mmy=0:my-1; mmz=shiftdim(0:mz-1,-1);
963     le = mod(mmx+mod(nx-2,mx),mx)+1;
964     patches.le = reshape( le+mx*(mmy+my*mmz) ,[],1);
965     ri = mod(mmx-mod(nx-2,mx),mx)+1;
966     patches.ri = reshape( ri+mx*(mmy+my*mmz) ,[],1);
967     bo = mod(mmy+mod(ny-2,my),my)+1;
968     patches.bo = reshape( 1+mmx+mx*(bo-1+my*mmz) ,[],1);
969     to = mod(mmy-mod(ny-2,my),my)+1;
970     patches.to = reshape( 1+mmx+mx*(to-1+my*mmz) ,[],1);
971     ba = mod(mmz+mod(nz-2,mz),mz)+1;
972     patches.ba = reshape( 1+mmx+mx*(mmy+my*(ba-1)) ,[],1);
973     fr = mod(mmz-mod(nz-2,mz),mz)+1;
974     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.

[5](#) [6](#)

```

983 if prod(ratio)*patches.nEnsem>0.9, warning( ...
984 'Probably poor scale separation in ensemble of coupled phase-shifts')
985 scaleSeparationParameter = ratio*patches.nEnsem
986 end

```

End the two if-statements.

```

992 end%if-else nEnsem>1
993 end%if not-empty(cs)

```

<sup>5</sup> **ToDo:** Need to justify this and the arbitrary threshold more carefully??

<sup>6</sup> **ToDo:** Is this ratio appropriate???? or patches.ratio??

**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>7</sup>

```
1012 if patches.parallel
1013     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.

```
1022 [~,pari]=max(nPatch+0.01*(1:3));
1023 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.

```
1033 switch pari
1034     case 1, patches.x=codistributed(patches.x,patches.codist);
1035     case 2, patches.y=codistributed(patches.y,patches.codist);
1036     case 3, patches.z=codistributed(patches.z,patches.codist);
1037 otherwise
1038     error('should never have bad index for parallel distribution')
1039 end%switch
1040 end%spmd
```

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

```
1048 else% not parallel
1049     if isfield(patches,'codist'), rmfield(patches,'codist'); end
1050 end%if-parallel
```

## Fin

```
1059 end% function
```

---

<sup>7</sup> If subsequently outside `spmd`, then one must use functions like `getfield(patches{1}, 'a')`.

## 5.2 patchSys3(): interface 3D space to time integrators

To simulate in time with 3D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. Communicate patch-design variables (Section 5.1) either via the global struct `patches` or via an optional third argument. `patches` is required for the parallel computing of `spmd`, or if parameters are to be passed though to the user microscale function.

```
23 function dudt = patchSys3(t,u,patches,varargin)
24 if nargin<3, global patches, end
```

### Input

- `u` is a vector/array of length  $\text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$  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  $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times \text{nPatch}(2) \times 1$  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 list 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 `prod(nSubP) · nVars · nEnsem · prod(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()`.

```
104 sizeu = size(u);
105 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.

```
116 dudt = patches.fun(t,u,patches,varargin{:});
117 m = patches.nEdge(1);
118 dudt([1:m end-m+1:end],:,:,:,:) = 0;
119 m = patches.nEdge(2);
120 dudt(:,:,1:m end-m+1:end,:,:) = 0;
121 m = patches.nEdge(3);
122 dudt(:,:,1:m end-m+1:end,:,:) = 0;
123 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.<sup>8</sup>

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.

---

<sup>8</sup> Script `patchEdgeInt3test.m` verifies most of 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).
- `.nEdge`, three elements, the width of edge values set by interpolation at the  $x, y, z$ -face regions, respectively, of each patch (default is one all  $x, y, z$ -faces).
- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

## Output

- `u` is 8D array,  $nSubP1 \cdot nSubP2 \cdot nSubP3 \cdot nVars \cdot nEnsem \cdot nPatch1 \cdot nPatch2 \cdot nPatch3$ , of the fields with face values set by interpolation.

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

```

127 [~,~,nz,~,~,~,~,Nz] = size(patches.z);
128 [~,ny,~,~,~,~,Ny,~] = size(patches.y);
129 [nx,~,~,~,~,Nx,~,~] = size(patches.x);
130 nEnsem = patches.nEnsem;
131 nVars = round( numel(u)/numel(patches.x) ...
132     /numel(patches.y)/numel(patches.z)/nEnsem );
133 assert(numel(u) == nx*ny*nz*Nx*Ny*Nz*nVars*nEnsem ...
134     , 'patchEdgeInt3: input u has wrong size for parameters')
135 u = reshape(u,[nx ny nz nVars nEnsem Nx Ny Nz]);

```

**Implement multiple width edges by folding** Subsample  $x, y, z$  coordinates, noting it is only differences that count *and* the microgrid  $x, y, z$  spacing must be uniform.

```

146 x = patches.x;
147 y = patches.y;
148 z = patches.z;
149 if mean(patches.nEdge)>1
150     mx = patches.nEdge(1);
151     my = patches.nEdge(2);
152     mz = patches.nEdge(3);
153     x = x(1:mx:nx,:,:,:, :, :, :, :);
154     y = y(:,1:my:ny,:,:,:, :, :, :, :);
155     z = z(:, :, 1:mz:nz,:,:,:, :, :, :, :);
156     nx = nx/mx;
157     ny = ny/my;
158     nz = nz/mz;

```

```

159     u = reshape(u,mx,nx,my,ny,mz,nz,nVars,nEnsem,Nx,Ny,Nz);
160     nVars = nVars*mx*my*mz;
161     u = reshape( permute(u,[2:2:6 1:2:5 7:11]) ...
162                 ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
163 end%if patches.nEdge

```

**Staggered grid** 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.

```

176 if patches.stag % transform by doubling the number of fields
177 error('staggered grid not yet implemented????')
178 v=nan(size(u)); % currently to restore the shape of u
179 u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
180 stagShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
181 iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
182 r=r/2; % ratio effectively halved
183 nPatch=nPatch/2; % halve the number of patches
184 nVars=nVars*2; % double the number of fields
185 else % the values for standard spectral
186     stagShift = 0;
187     iV = 1:nVars;
188 end%if patches.stag

```

### 5.3.1 Interpolate over the three successive directions

Only use the interior values of the fields for interpolating to the edges.

```
199 u = u(2:nx-1,2:ny-1,2:nz-1,:,:,:,:,:);
```

Interpolate in turn, the edge or mid-patch faces normal to the  $x, y, z$ -directions, in this way we naturally fill-in face-edge and corner values.

```

207 u = patchEdgeIntCore(1,u,x,patches,stagShift ...
208     ,1,nx,(ny-2)*(nz-2)*nVars,nEnsem,1,Nx,Ny*Nz ...
209     ,patches.le,patches.ri);
210 u = patchEdgeIntCore(2,u,y,patches,stagShift ...
211     ,nx,ny,(nz-2)*nVars,nEnsem,Nx,Ny,Nz ...
212     ,patches.bo,patches.to);
213 u = patchEdgeIntCore(3,u,z,patches,stagShift ...
214     ,nx*ny,nz,    nVars,nEnsem,Nx*Ny,Nz,1 ...
215     ,patches.ba,patches.fr);

```

Restore array  $u$  to its original shape.

```
221 u = reshape(u,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz);
```

**Unfold multiple edges** No need to restore  $x, y, z$ .

```

232 if mean(patches.nEdge)>1
233     nVars = nVars/(mx*my*mz);

```

```
234     u = reshape( u ,nx,ny,nz,mx,my,mz,nVars,nEnsem,Nx,Ny,Nz) ;
235     nx = nx*mx;
236     ny = ny*my;
237     nz = nz*mz;
238     u = reshape( permute(u,[4 1 5 2 6 3 7:11]) ...
239                 ,nx,ny,nz,nVars,nEnsem,Nx,Ny,Nz) ;
240 end%if patches.nEdge
Fin, returning the 8D array of field values with interpolated faces.
248 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.

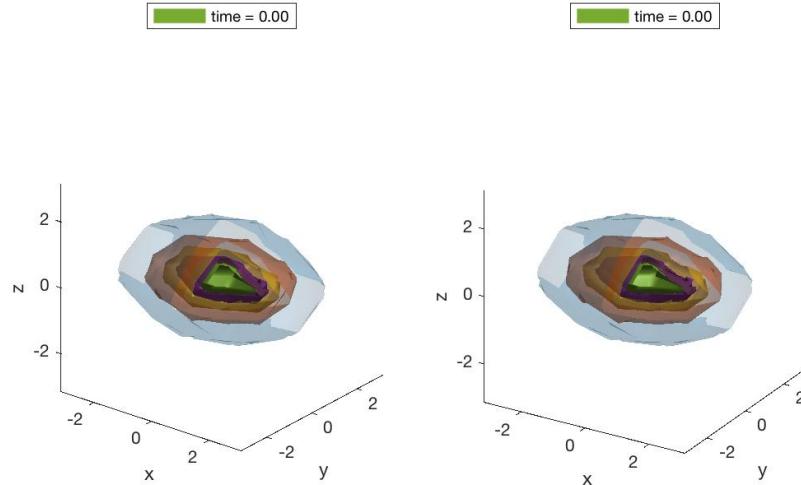
```
76 if ~exist('OCTAVE_VERSION','builtin')
77 [ts,us] = ode23(@patchSys3, 0.3*linspace(0,1,50).^2, u0(:));
78 else % octave version
79 [ts,us] = odeOcts(@patchSys3, 0.3*linspace(0,1).^2, u0(:));
80 end
```

**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

```

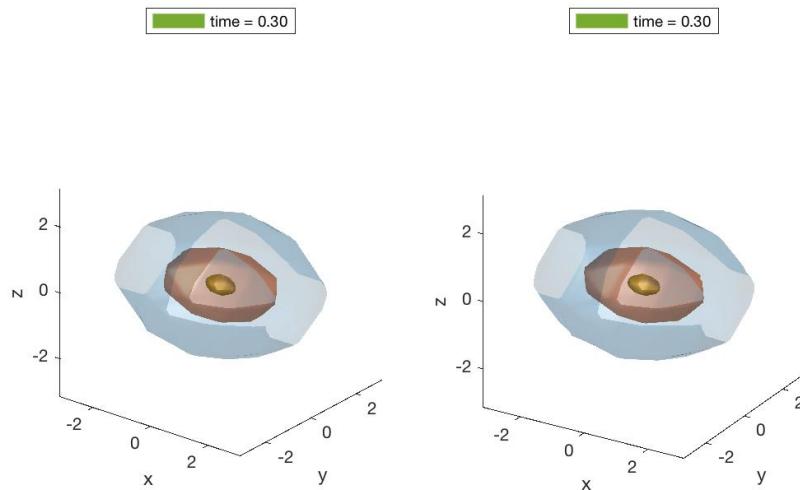
Find which elements of the 8D array are interior micro-grid points and hence correspond to dynamical variables.

```

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
230     ord=ord

```

Configure with same heterogeneity.

```

236 configPatches3(@heteroDiff3, [-pi pi], nan, nPatch ...
237 , ord, ratio, nSubP, 'EdgyInt', true, 'nEnsem', nEnsem ...
238 , 'hetCoeffs', cHetr);

```

Construct the Jacobian of the scheme as the matrix of the linear transformation, obtained by transforming the standard unit vectors.

```

246 jac = nan(length(i));
247 for j = 1:length(i)

```

```

248     u = u0(:)+(i(j)==(1:numel(u0))');
249     tmp = patchSys3(0,u);
250     jac(:,j) = tmp(i);
251 end

```

Test for symmetry, with error if we know it should be symmetric.

```

258 notSymmetric=norm(jac-jac')
259 % if notSymmetric>1e-7, spy(abs(jac-jac')>1e-7), end%??
260 assert(notSymmetric<1e-7,'failed symmetry')

```

Find all the eigenvalues (as `eigs` is unreliable), and put eigenvalues in a vector.

```

267 [evecs,evals] = eig((jac+jac')/2,'vector');
268 biggestImag=max(abs(imag(evals)));
269 if biggestImag>0, biggestImag=biggestImag, end

```

Sort eigenvalues on their real-part with most positive first, and most negative last. Store the leading eigenvalues in `egs`, and write out when computed all orders. The number of zero eigenvalues, `nZeroEv`, gives the number of decoupled systems in this patch configuration.

```

279 [~,k] = sort(-real(evals));
280 evals=evals(k); evecs=evecs(:,k);
281 if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
282 % if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
283 % else % find evec closest to that of each leading spectral
284 %       [~,k]=max(abs(evecs'*evec0));
285 % evals=evals(k); % re-sort in corresponding order
286 % end
287 leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
288 end
289 disp(' spectral quadratic quartic sixth-order ...')
290 leadingEvals=leadingEvals

```

#### 5.4.3 `heteroDiff3()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSys3`, [Section 5.2](#)), computes the time derivative [\(3.1\)](#) at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities,  $c_{ijk}^x$ ,  $c_{ijk}^y$  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

```

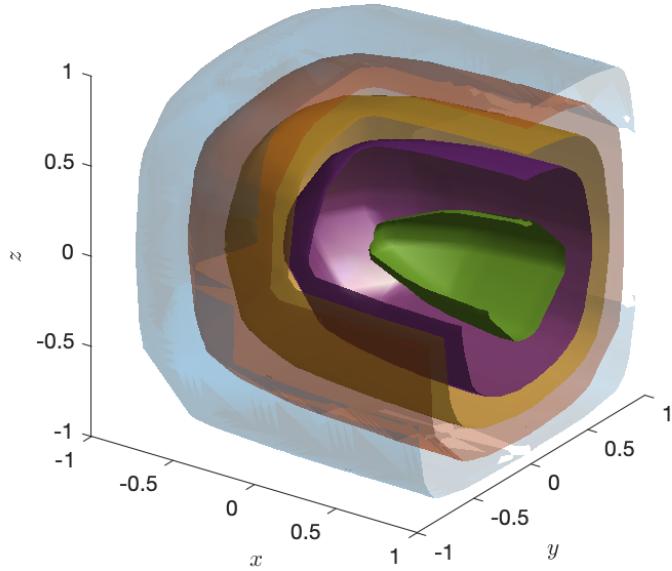
```
33     dz = diff(patches.z(2:3)); % z micro-scale step
34     i = 2:size(u,1)-1; % x interior points in a patch
35     j = 2:size(u,2)-1; % y interior points in a patch
36     k = 2:size(u,3)-1; % z interior points in a patch
```

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 3.19](#)).

```

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 heteroDispersiveWave3: heterogeneous Dispersive Waves from 4th order PDE

This uses small spatial patches to simulate heterogeneous dispersive waves in 3D. The wave equation for  $u(x, y, z, t)$  is the fourth-order in space PDE

$$u_{tt} = -\nabla^2(C\nabla^2u)$$

for microscale variations in scalar  $C(x, y, z)$ .

Initialise some Matlab aspects.

```
21 clear all
22 cMap=jet(64); cMap=0.8*cMap(7:end-7,:); % set colormap
23 basename = [num2str(floor(1e5*rem(now,1))) mfilename]
24 %global OurCf2eps, OurCf2eps=true %optional to save plots
```

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the macro-wave time scale is roughly one.

```
34 mPeriod = [2 2 2];
35 cHetr = exp(0.9*randn(mPeriod));
36 cHetr = cHetr*mean(1./cHetr(:))
```

Establish global patch data struct to interface with a function coding a fourth-order heterogeneous wave PDE: to be solved on  $[-\pi, \pi]^3$ -periodic domain, with  $5^3$  patches, spectral interpolation (0) couples the patches, each patch with micro-grid spacing 0.22 (relatively large for visualisation), and with  $6^3$  points forming each patch. (Six because two edge layers on each of two faces, and two interior points for the PDE.)

```
50 global patches
51 patches = configPatches3(@heteroDispWave3, [-pi pi] ...
52 , 'periodic', 5, 0, 0.22, mPeriod+4 , 'EdgyInt', true ...
53 , 'hetCoeffs', cHetr , 'nEdge', 2);
```

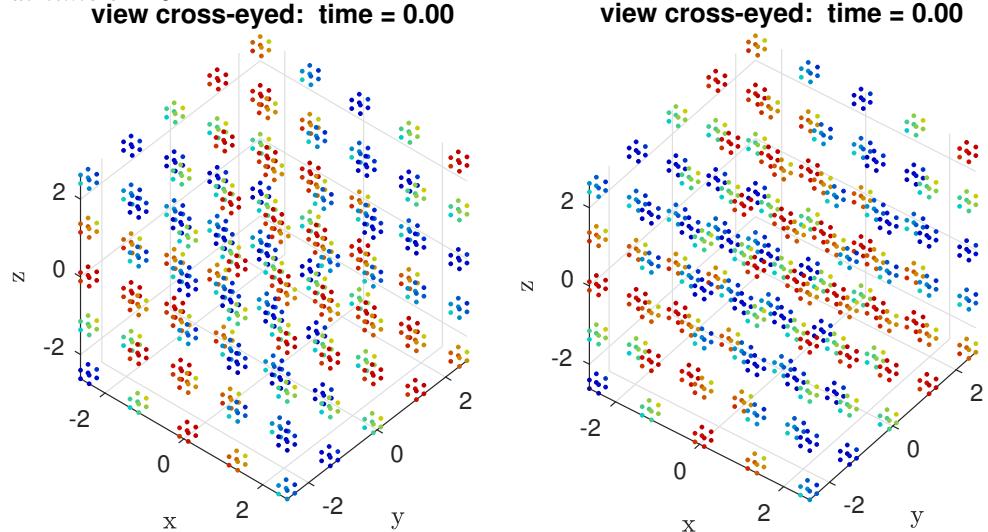
Set a wave initial state using auto-replication of the spatial grid, and as [Figure 5.6](#) shows. This wave propagates diagonally across space. Concatenate the two  $u, v$ -fields to be the two components of the fourth dimension.

```
64 u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
65 v0 = -0.5*cos(patches.x+patches.y+patches.z)*3;
66 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).

```
83 disp('Simulate heterogeneous wave u_tt=delsq[C*delsq(u)]')
84 tic
85 [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
86 simulateTime=toc
```

*Figure 5.6: initial field  $u(x, y, z, t)$  at time  $t = 0$  of the patch scheme applied to a heterogeneous dispersive wave PDE: [Figure 5.7](#) plots the computed field at time  $t = 6$ .*



Animate the computed simulation to end with [Figure 5.7](#). Use `patchEdgeInt3` to obtain patch-face values in order to most easily reconstruct the array data structure.

Replicate  $x$ ,  $y$ , and  $z$  arrays to get individual spatial coordinates of every data point. Then, optionally, set faces to `nan` so the plot just shows patch-interior data.

```

100 %
101 figure(1), clf, colormap(cMap)
102 xs = patches.x+0*patches.y+0*patches.z;
103 ys = patches.y+0*patches.x+0*patches.z;
104 zs = patches.z+0*patches.y+0*patches.x;
105 if 1, xs([1:2 end-1:end],:,:, :)=nan;
106     xs(:, [1:2 end-1:end],:,:, :)=nan;
107     xs(:,:, [1:2 end-1:end],:)=nan;
108 end;%option
109 j=find(~isnan(xs));

```

In the scatter plot, `col()` maps the  $u$ -data values to the colour of the dots.

```
116 col = @ (u) sign(u).*abs(u);
```

Loop to plot at each and every time step.

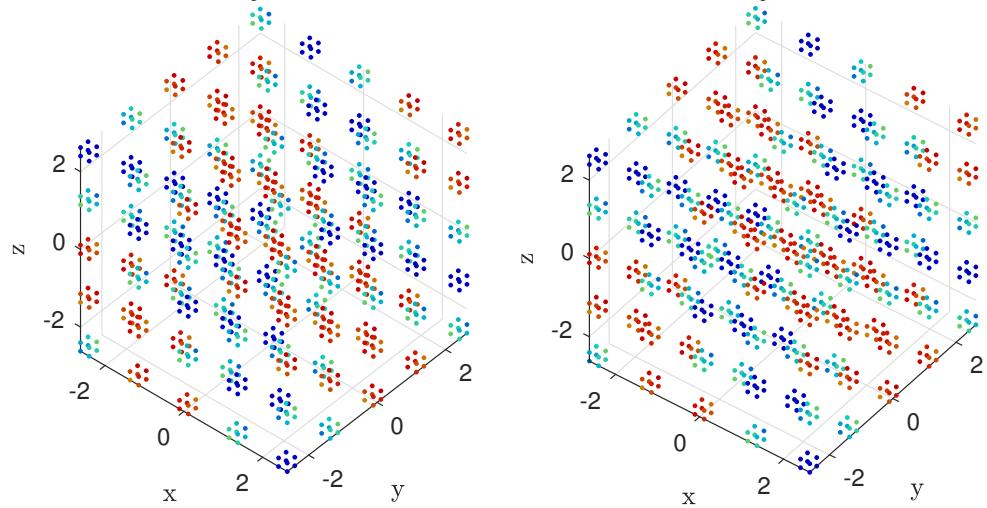
```

122 for i = 1:length(ts)
123     uv = patchEdgeInt3(us(i,:));
124     u = uv(:,:, :, 1,:);
125     for p=1:2
126         subplot(1,2,p)
127         if (i==1)
128             scat(p) = scatter3(xs(j),ys(j),zs(j),'.');

```

Figure 5.7: field  $u(x, y, z, t)$  at time  $t = 6$  of the patch scheme applied to the heterogeneous dispersive wave PDE with initial condition in Figure 5.6.

**view cross-eyed: time = 6.00**      **view cross-eyed: time = 6.00**



```

129      axis equal, caxis(col([0 1])), view(45-4*p,42)
130      xlabel('$x$'), ylabel('$y$'), zlabel('$z$')
131    end
132    title(['view cross-eyed: time = ' num2str(ts(i),'%4.2f')])
133    set( scat(p),'CData',col(u(j)) );
134  end

```

Optionally save the initial condition to graphic file for Figure 5.6, and optionally save the last plot.

```

142  if i==1,
143    ifOurCf2eps([basename 'ic'])
144      disp('Type space character to animate simulation')
145      pause
146    else pause(0.1)
147    end
148  end% i-loop over all times
149  ifOurCf2eps([basename 'fin'])

```

### 5.6.1 heteroDispWave3(): PDE function of 4th-order heterogeneous dispersive waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE for  $u(x, y, z, t)$  and ‘velocity’  $v(x, y, z, t)$  is

$$u_t = v, \quad v_t = -\nabla^2(C\nabla^2 u)$$

for microscale variations in scalar  $C(x, y, z)$ . For 8D input arrays  $\mathbf{u}$ ,  $\mathbf{x}$ ,  $\mathbf{y}$ , and  $\mathbf{z}$  (via edge-value interpolation of `patchSys3`, Section 5.2), computes the time derivative at each point in the interior of a patch, output in  $\mathbf{ut}$ . The 3D array of heterogeneous coefficients,  $C_{ijk}$ ,  $c_{ijk}^y$  and  $c_{ijk}^z$ , have been stored in `patches.cs` (3D).

Supply patch information as a third argument (required by parallel computation), or otherwise by a global variable.

```

187 function ut = heteroDispWave3(t,u,patches)
188   if nargin<3, global patches, end
190
191   Micro-grid space steps.
192
193   dx = diff(patches.x(2:3));
194   dy = diff(patches.y(2:3));
195   dz = diff(patches.z(2:3));

```

First, compute  $C\nabla^2 u$  into say  $u$ , using indices for all but extreme micro-grid points. We use a single colon to represent the last four array dimensions because the result arrays are already dimensioned.

```

205 I = 2:size(u,1)-1; J = 2:size(u,2)-1; K = 2:size(u,3)-1;
206 u(I,J,K,1,:) = patches.cs(I,J,K,1,:).*(
207   diff(u(:,J,K,1,:),2,1)/dx^2 ...

```

Reserve storage, set lowercase indices to non-edge interior, and then assign interior patch values to the heterogeneous diffusion time derivatives.

```

215 ut = nan+u; % preallocate output array
216 i = I(2:end-1); j = J(2:end-1); k = K(2:end-1);
217 ut(i,j,k,1,:) = u(i,j,k,2,:); % du/dt=v
218 % dv/dt=delta^2 of above C*delta^2
219 ut(i,j,k,2,:) = -( diff(u(I,j,k,1,:),2,1)/dx^2 ...
220   +diff(u(i,J,k,1,:),2,2)/dy^2 +diff(u(i,j,K,1,:),2,3)/dz^2 );
221 end% function

```

### 5.6.2 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, but beware that some realisations take several minutes.

```

21 clear all, close all
22 global patches
23 nRealise = 10

```

#### 5.6.2.1 Check divided difference interpolation

```

36 fprintf('\n\n**** Check divided difference interpolation\n')
37 pause(1)

```

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.)

```

46 maxErrors=[];
47 for realisation = 1:nRealise
48     nEdge = randi(3,1,3)% =1,2, or 3
49     edgyInt = (rand>0.5)
50     Lx = 1+3*rand; Ly = 1+3*rand; Lz = 1+3*rand;
51     xyzLim = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
52     nSubP = nEdge.*((2-edgyInt)*randi(2,1,3)+1+edgyInt )
53     ordCC = 2*randi(3)
54     nPatch = ordCC+randi(4,1,3)
55     dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
56     configPatches3(@sin,xyzLim,'equispace',nPatch,ordCC ...
57         ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

Second, displace patches to a random non-uniform spacing.

```

63 Hx = diff(patches.x(1,1,1,:,:,:1:2,1,1));
64 patches.x = patches.x+0.8*Hx*(rand(1,1,1,1,1,nPatch(1),1,1)-0.5);
65 Hx = squeeze( diff(patches.x(1,1,1,:,:,:,1,1)) );% for information only
66 Hy = diff(patches.y(1,1,1,:,:,:,1,1:2,1));
67 patches.y = patches.y+0.8*Hy*(rand(1,1,1,1,1,1,nPatch(2),1)-0.5);
68 Hy = squeeze( diff(patches.y(1,1,1,:,:,:,1,:1)) );% for information only
69 Hz = diff(patches.z(1,1,1,:,:,:,1,1:2));
70 patches.z = patches.z+0.8*Hz*(rand(1,1,1,1,1,1,nPatch(3))-0.5);
71 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.

```

85 ox=min(ordCC,nPatch(1)-1);
86 oy=min(ordCC,nPatch(2)-1);
87 oz=min(ordCC,nPatch(3)-1);
88 [ps,qs,rs]=ndgrid(0:ox,0:oy,0:oz);
89 ps=reshape(ps,1,1,1,[ ]);
90 qs=reshape(qs,1,1,1,[ ]);
91 rs=reshape(rs,1,1,1,[ ]);
92 cs=2*rand(size(ps))-1;
93 u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;

```

Then evaluate the interpolation, setting faces to **inf** for error checking.

```

100 u=u0;
101 u([1:nEdge(1) end-nEdge(1)+1:end],:,:,:)=inf;
102 u(:,[1:nEdge(2) end-nEdge(2)+1:end],:,:)=inf;
103 u(:,:,1:nEdge(3) end-nEdge(3)+1:end],:,:)=inf;
104 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.

```

114     error = ui-u0;
115     hist(log10(abs(error(abs(error)>1e-20))),-20:-6)
116     xlabel('log10 error'), pause(0.3)%??
117     maxError=max(abs(error(:)))
118     maxErrors=[maxErrors maxError];
119     assert(maxError<1e-10*4^ordCC ...
120         , 'failed divided difference interpolation')
121     disp('*** This divided difference test passed')

```

End the for-loops over various parameters.

```

128 end% for realisation
129 maxMaxErrorDividedDiffs = max(maxErrors)
130 disp('***** Passed all divided difference interpolation')
131 pause(1)

```

#### 5.6.2.2 Test standard spectral interpolation

```

141 fprintf('\n\n***** Test standard spectral interpolation\n')
142 pause(1)

```

Test over various numbers of patches, random domain lengths and random ratios. Try realisations of random tests.

```
149 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.

```

157 nEdge=randi(3,1,3)% =1,2, or 3
158 edgyInt = (rand>0.5)
159 Lx = 1+3*rand, Ly = 1+3*rand, Lz = 1+3*rand
160 xyzLim = [0 Lx 0 Ly 0 Lz]-[rand*[1 1] rand*[1 1] rand*[1 1]]
161 nSubP = nEdge.*((2-edgyInt)*randi(3,1,3)+1+edgyInt )
162 nPatch = randi([3 6],1,3)
163 dx = [Lx Ly Lz]./nPatch./nSubP.*rand(1,3)/2
164 configPatches3(@sin,xyzLim,'periodic',nPatches,0 ...
165     ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

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.

```

175 nV=randi(3)
176 [nx,Nx]=size(squeeze(patches.x));
177 [ny,Ny]=size(squeeze(patches.y));
178 [nz,Nz]=size(squeeze(patches.z));
179 u0=nan(nx,ny,nz,nV,1,Nx,Ny,Nz);
180 for iV=1:nV

```

```

181     ks=floor( rand(1,3).*floor(([Nx Ny Nz]+1)/2) )
182     phis = pi*rand(1,3).*([2*ks]=[Nx Ny Nz])
183     % generate 8D array via auto-replication
184     u0(:,:,:,:,iV,1,:,:,:)=sin(2*pi*ks(1)*patches.x/Lx+phis(1)) ...
185                 .*sin(2*pi*ks(2)*patches.y/Ly+phis(2)) ...
186                 .*sin(2*pi*ks(3)*patches.z/Lz+phis(3));
187 end

```

Copy and nan the faces, then interpolate

```

193 u=u0;
194 u([1:nEdge(1) end-nEdge(1)+1:end],:,:,:)=nan;
195 u(:,[1:nEdge(2) end-nEdge(2)+1:end],:,:,:)=nan;
196 u(:,:,1:nEdge(3) end-nEdge(3)+1:end,:)=nan;
197 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.

```

206 error = u-u0;
207 assert(all(~isnan(error(:))), 'found nans in the error!')
208 hist(log10(abs(error(abs(error)>1e-20))), -20:-6)
209 xlabel('log10 error'), pause(0.3)%??
210 normError=norm(error(:))
211 assert(normError<1e-10, '3D spectral interpolation failed')
212 disp('*** This spectral test passed')

```

End the for-loop over realisations

```

219 end
220 disp('***** All the spectral tests passed')
221 pause(1)

```

### 5.6.2.3 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.)

```

239 for realisations=1:nRealise
240     nEdge = randi(3,1,3)% =1,2, or 3
241     edgyInt = (rand>0.5)
242     nSubP = nEdge.*((2-edgyInt)*randi(2,1,3)+1+edgyInt)
243     ordCC = 2*randi(3)
244     nPatch = ordCC+randi(3,1,3)
245     xyzLim=5*[-rand(1,3); rand(1,3)]
246     dx = diff(xyzLim)./nPatch./nSubP.*rand(1,3)/2
247     configPatches3(@sin,xyzLim,'periodic',nPatch,ordCC ...
248                 ,dx,nSubP,'EdgyInt',edgyInt,'nEdge',nEdge);

```

**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.

```

258     [ps,qs,rs]=meshgrid(0:ordCC);
259     ps=reshape(ps,1,1,1,[]);
260     qs=reshape(qs,1,1,1,[]);
261     rs=reshape(rs,1,1,1,[]);
262     cs=2*rand(size(ps))-1;
263     u0=cs.*patches.x.^ps.*patches.y.^qs.*patches.z.^rs;

```

Then evaluate the interpolation.

```
269     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.

```

277     I=ordCC/2+1:nPatch(1)-ordCC/2;
278     J=ordCC/2+1:nPatch(2)-ordCC/2;
279     K=ordCC/2+1:nPatch(3)-ordCC/2;
280     error=ui(:,:,(:,:,I,J,K))-u0(:,:,(:,:,I,J,K));
281     assert(all(~isnan(error(:))), 'found nans in the error!')
282     hist(log10(abs(error(abs(error)>1e-20))), -20:-6)
283     xlabel('log10 error'), pause(0.3)%%
284     normError=norm(error(:))
285     assert(normError<5e-8 ...
286         , 'failed finite stencil polynomial interpolation')
287     disp('*** This finite stencil test passed')

```

End the for-loops over various parameters.

```

294 end%for realisation
295 disp('***** Passed all polynomial interpolation tests')

```

#### 5.6.2.4 Finished

If no error messages, then all OK.

```
309 disp('***** All types of interpolation tests successful')
```

## 5.7 To do for patches

- Core averages code.
- Some staggered grid patches—although seems unnecessary.
- Adapt to maps in micro-time? Surely easy, just an example.
- Test hierarchy of patches.

---

## 6 Matlab parallel computation of the patch scheme

---

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.<sup>1</sup> The reason is that the patch scheme ([Chapter 3](#)) has a clear domain decomposition of assigning relatively few patches to each processor.

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](#)).

---

<sup>1</sup>This parallelisation is not written for, nor tested for, Octave.

## 6.1 chanDispSpmd: simulation of a 1D shear dispersion via simulation on small patches across a channel

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  $\text{Pe}^2$ —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  $\text{Pe}$  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  $\delta x$ , 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
```

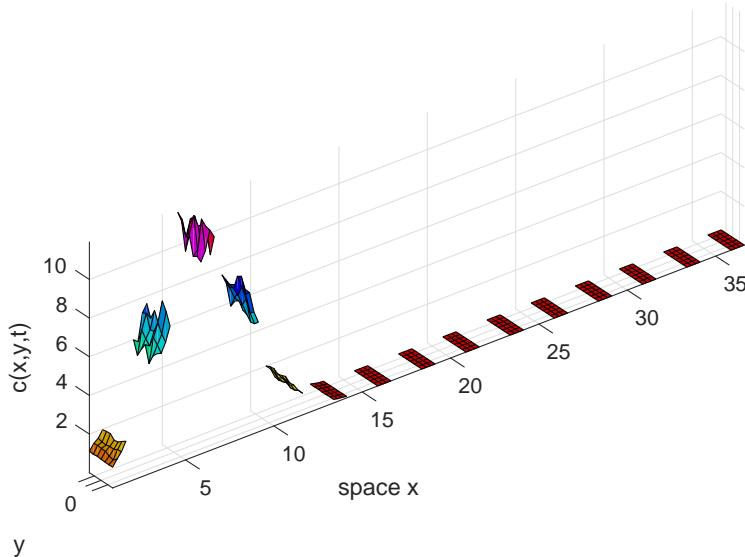
### 6.1.1 Simulate heterogeneous advection-diffusion

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.

*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.*



```

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

```

- 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)]);

```

- 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};

```

- 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};

```

- 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')])
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- $x$ .

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  $\exp[\mathcal{N}(0, 1)]$ .

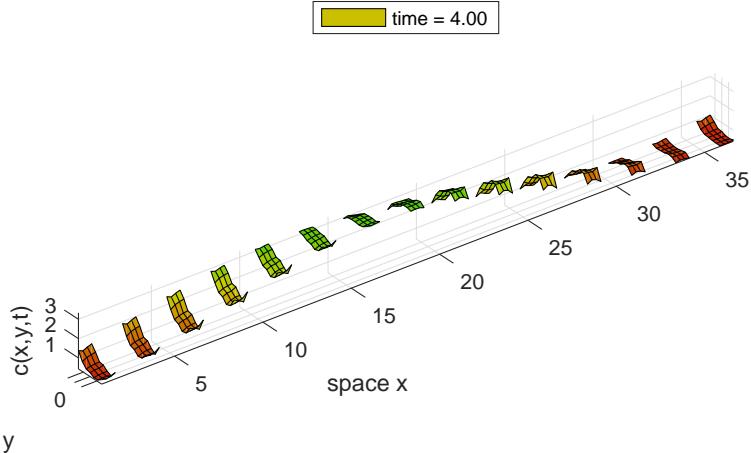
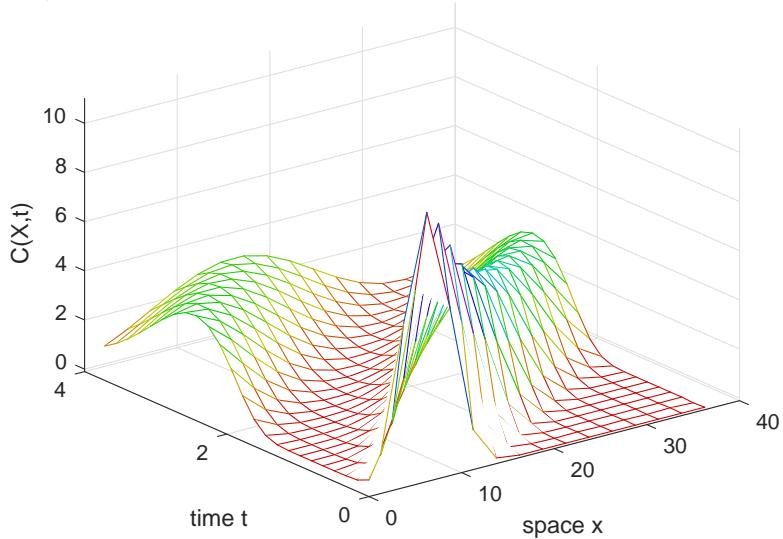


Figure 6.3: macro-scale view of heterogeneous advection-diffusion PDE along a (periodic) channel obtained via the patch scheme.



```

374 figure(2), clf, colormap(0.8* hsv)
375 X = squeeze(mean(x(2:end-1,:,:,:)));
376 C = squeeze(mean(mean(cs(:,2:end-1,:,:,:),2),3));
377 j = 1:ceil(nTimes/30):nTimes;
378 mesh(X,ts(j),C(j,:));
379 xlabel('space $x$'), ylabel('time $t$'), zlabel('$C(X,t)$')
380 zlim([-0.1 11])
381 ifOurCf2eps([mfilename 'Macro'])

```

### 6.1.3 microBurst function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.2. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take

take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0033. With macro-step 0.5, these parameters usually give stable projective integration.

```

404 function [tbs,xbs] = aBurst(tb0,xb0,patches)
405     normx=max(abs(xb0(:)));
406     disp(['* aBurst t=' num2str(tb0) ' |x|=' num2str(normx)])
407     assert(normx<20,'solution exploding')
408     tbs = tb0+(0:0.033:0.2);
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 advectons 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

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

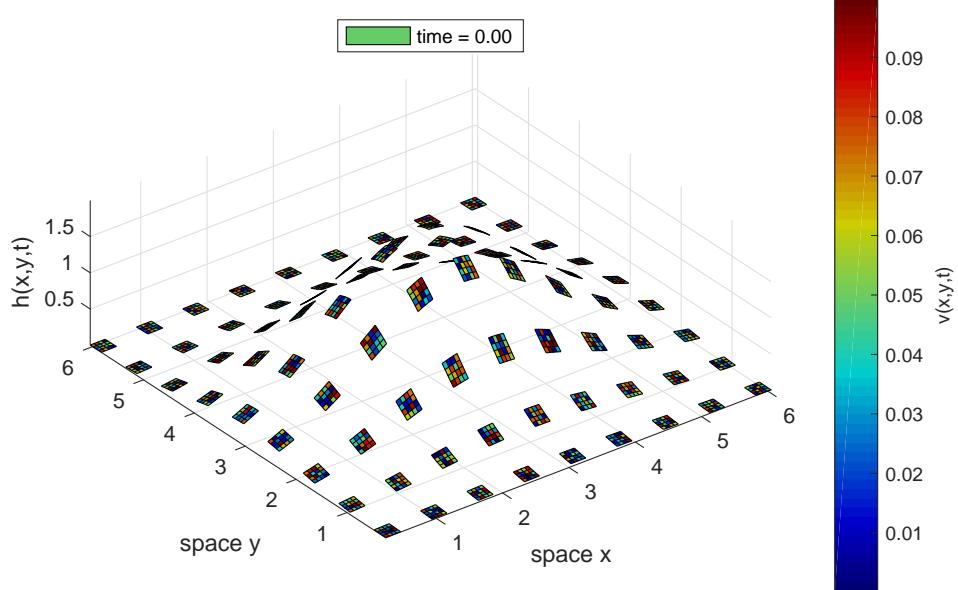
Set initial conditions of a simulation as shown in [Figure 6.4](#). Here the initial condition is a (periodic) quasi-Gaussian in  $h$  and zero velocity  $\vec{v}$ , with additive random perturbations.

```
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));
```

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.



```

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)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
235 case 2
236 huvs = RK2mesoPatch(ts,huv0);
237 huvs = huvs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
246 case 3,spmd
247 huvs = RK2mesoPatch(ts,huv0,[],patches);
248 end%spmd
249 huvs = huvs{1};
```

4. In this fourth case, use Projective Integration (PI). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` ([Section 6.2.3](#)) to compute each burst of the micro-scale simulation. The macro-scale time-step needs to be less than about 0.1 (which here is not much projection). The function `microBurst()` interfaces to `aBurst()` ([Section 6.2.3](#)) in order to provide shaped initial states, and to provide the patch information.

```
264 case 4
265 microBurst = @(tb0,xb0,bT) ...
266 aBurst(tb0 ,reshape(xb0,size(huv0)) ,patches);
267 ts = 0:0.1:1
268 huvs = PIRK2(microBurst,ts,gather(huv0(:)));
269 huvs = reshape(huvs,[length(ts) size(huv0)]);
```

End the four cases.

```
276 end%switch theCase
```

## 6.2.2 Plot the solution

Optionally set to save some plots to file.

```
287 if 0, global OurCf2eps, OurCf2eps=true, end
```

### Animate the computed solution field over time

```
293 figure(1), clf, colormap(0.8*jet)
```

First get the  $x$ -coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
300 if theCase==1, x = patches.x;
301         y = patches.y;
302 else, spmd
303     x = gather( patches.x );
304     y = gather( patches.y );
305 end%spmd
306 x = x{1}; y = y{1};
307 end
308 x([1 end],:,:,:,:,:) = nan;
309 y(:,[1 end],:,:,:,:,:) = nan;
```

Draw the field values as a patchy surface evolving over 100–200 time steps.

```
316 nTimes = length(ts)
317 for l = 1:ceil(nTimes/200):nTimes
```

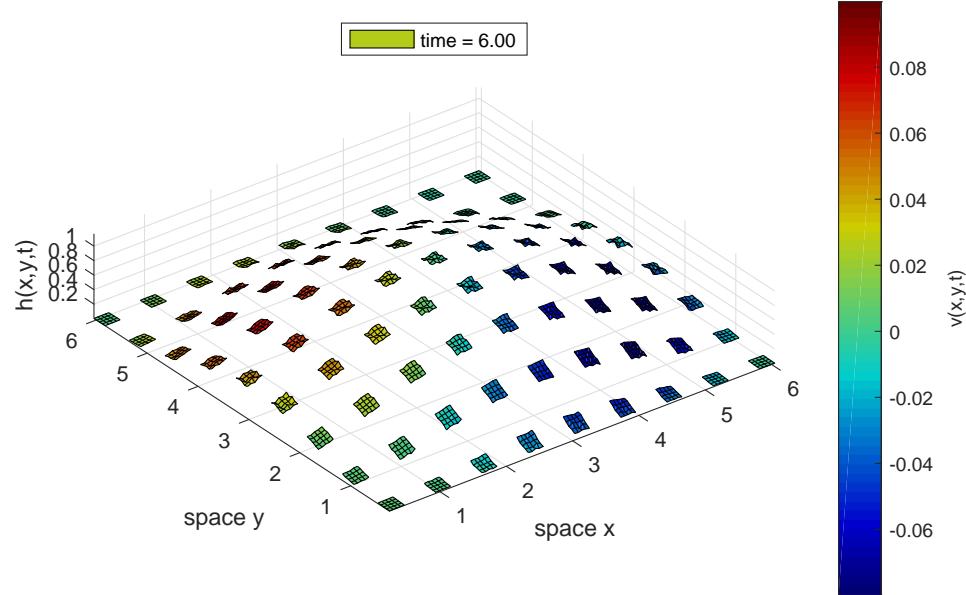
At each time, squeeze sub-patch data fields into three 4D arrays, permute to get all the  $x/y$ -variations in the first/last two dimensions, and then reshape to 2D.

```
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'));
340     axis([0 Len 0 Len 0 max(h(:))])
341     c = colorbar; c.Label.String = 'v(x,y,t)';
342     legend(['time = ' num2str(ts(l),'%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(l),'%4.2f')])
```

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  $\exp[\mathcal{N}(0, 1)]$ .



```

352     pause(0.1)
353 end
370 end%for over time
371 if0urCf2eps([mfilename 'tFin'])

```

### 6.2.3 microBurst function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.01. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take 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  $\nu_{ij}$ , have previously been merged and stored in the array `patches.cs` (2D  $\times$  3): herein `patches` is named `p`.

```

24 function huvt = rotFilmMicro(t,huv,p)
25 [nx,ny,~]=size(huv); % micro-grid points in patches
26 i = 2:nx-1;           % x interior points in a patch
27 j = 2:ny-1;           % y interior points in a patch
28 dx = diff(p.x(2:3)); % x space step
29 dy = diff(p.y(2:3)); % y space step
30 huvt = nan+huv;      % preallocate output array

```

Set indices of fields in the arrays. Need to store different diffusivity values for the  $x, y$ -directions as they are evaluated at different points in space.

```

38 h=1; u=2; v=3;
39 b=1; nux=2; nuy=3;

```

Use a staggered micro-grid so that  $h(i,j) = h_{ij}$ ,  $u(i,j) = u_{i+1/2,j}$ , and  $v(i,j) = v_{i,j+1/2}$ . We need the following to interpolate some quantities to other points on the staggered micro-grid. But the first two statements fill-in two needed corner values because they are not (currently) interpolated by `patchEdgeInt2()`.

```

51 huv(1,ny,u,:,:,:) = huv(2,ny,u,:,:,:)+huv(1,ny-1,u,:,:,:);
52                               -huv(2,ny-1,u,:,:,:);
53 huv(nx,1,v,:,:,:) = huv(nx,2,v,:,:,:)+huv(nx-1,1,v,:,:,:);
54                               -huv(nx-1,2,v,:,:,:);
55 v4u = (huv(i,j-1,v,:,:,:)+huv(i+1,j,v,:,:,:)) ...
56                               +huv(i,j,v,:,:,:)+huv(i+1,j-1,v,:,:,:))/4;
57 u4v = (huv(i,j+1,u,:,:,:)+huv(i-1,j,u,:,:,:)) ...
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

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
```

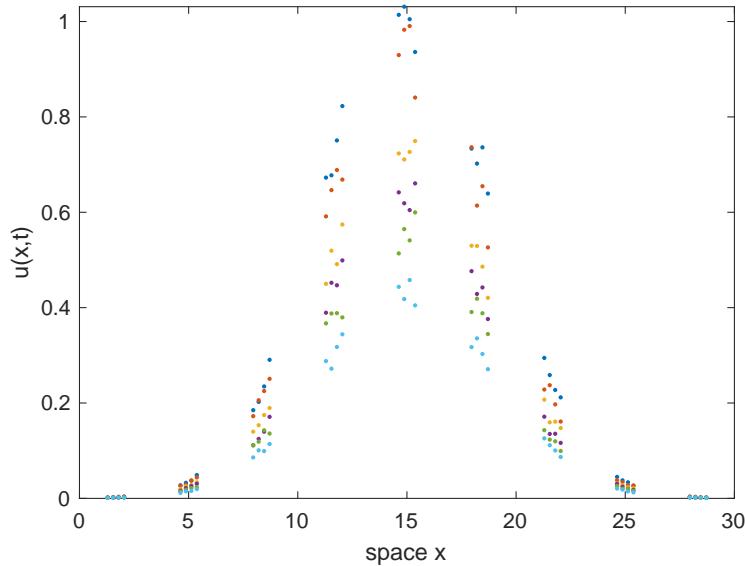
Set micro-scale heterogeneity with various spatial periods in the three directions.

```
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 ...
```

*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  $t = 0.4$ .*



```

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
```

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.<sup>2</sup> The function `microBurst()` here interfaces to `aBurst()` (Section 6.3.3) in order to provide shaped initial states, and to provide the patch information.

---

<sup>2</sup> Curiously, `PIG()` appears to suffer unrecoverable instabilities with its variable step size!

```

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)]);

```

End the four cases.

```

222 end%switch theCase

```

### 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),[] );

```

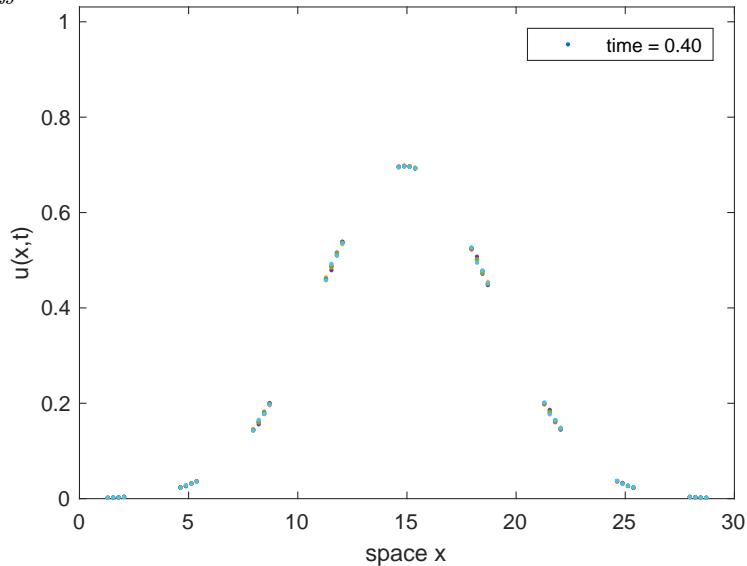
Draw point data to show spread at each cross-section, as well as macro-scale variation in the long space direction.

```

273 if l==1
274   hp = plot(x,u,'.');
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(l),'%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

```

Figure 6.7: final field  $u(x, y, z, 0.4)$  of the patch scheme applied to a heterogeneous diffusion PDE.



```

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 ifOrCf2eps([mfilename 'tFin'])

```

### 6.3.3 microBurst function for Projective Integration

Projective Integration stability seems to need bursts longer than 0.2. Here take ten meso-steps, each with default ten micro-steps so the micro-scale step is 0.002. With macro-step 3, these parameters usually give stable projective integration (but not always).

```

315 function [tbs,xbs] = aBurst(tb0,xb0,patches)
316     normx=max(abs(xb0(:)));
317     disp(['aBurst t = ', num2str(tb0), '|x| = ', num2str(normx)])
318     assert(normx<10,'solution exploding')
319     tbs = tb0+(0:0.02:0.2);
320     spmd
321         xb0 = codistributed(xb0,patches.codist);
322         xbs = RK2mesoPatch(tbs,xb0,[],patches);
323     end%spmd
324     xbs=reshape(xbs{1},length(tbs),[]);
325 end%function

```

Fin.

## 6.4 RK2mesoPatch()

This is a Runge–Kutta, 2nd order, integration of a given deterministic system of ODEs on patches. It invokes meso-time updates of the patch-edge values in order to reduce interpolation costs, and uses a linear variation in edge-values over the meso-time-step (Bunder et al. 2016, case  $Q = 2$ ). This function is aimed primarily for large problems executed on a computer cluster to markedly reduce expensive communication between computers.

If using within projective integration, it appears quite tricky to get all the time-steps chosen appropriately. One has to choose times for: the micro-scale time-step, the meso-time interval between communications, the longer meso-time burst length, and the macro-scale integration time-step.

```
27 function [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches)
28 if nargin<4, global patches, end
```

### Input

- `patches.fun()` is a function such as `dxdt=fun(t,x,patches)` that computes the right-hand side of the ODE  $d\vec{x}/dt = \vec{f}(t, \vec{x})$  where  $\vec{x}$  is a vector/array,  $t$  is a scalar, and the result  $\vec{f}$  is a correspondingly sized vector/array.
- `x0` is an vector/array of initial values at the time `ts(1)`.
- `ts` is a vector of meso-scale times to compute the approximate solution, say in  $\mathbb{R}^\ell$  for  $\ell \geq 2$ .
- `nMicro`, optional, default 10, is the number of micro-time-steps taken for each meso-scale time-step.
- `patches` struct set by `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  $\mathbb{R}^\ell$  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.<sup>3</sup>

```

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.

---

<sup>3</sup> 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/func1.m`, so that users simply invoke the function in MATLAB/Octave as usual by `func1(...)`.

Some editors may need to be told that `func1.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**

---

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. (2022), ‘High-order homogenisation by learning spatial discretisations of PDEs that provably preserve self-adjointness’, *Partial Differential Equations in Applied Mathematics* **6**(100449), 1–16.
- 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.
- Divahar, J., Roberts, A. J., Mattner, T. W., Bunder, J. E. & Kevrekidis, I. G. (2022), 'Two novel families of multiscale staggered patch schemes efficiently simulate large-scale, weakly damped, linear waves', *Computer Methods in Applied Mechanics and Engineering* accepted **16/5/2023**.
- 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.
- Gear, C. W. & Kevrekidis, I. G. (2005), 'Constraint-Defined Manifolds: a Legacy Code Approach to Low-Dimensional Computation', *Journal of Scientific Computing* **25**(1), 17–28.
- 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.
- Hu, Z., Pan, G., Wang, Y. & Wu, Z. (2016), 'Sparse principal component analysis via rotation and truncation', *IEEE Transactions on Neural Networks and Learning Systems* **27**(4), 875–890.
- Hyman, J. M. (2005), 'Patch dynamics for multiscale problems', *Computing in Science & Engineering* **7**(3), 47–53.  
<http://scitation.aip.org/content/aip/journal/cise/7/3/10.1109/MCSE.2005.57>
- Kevrekidis, I. G., Gear, C. W. & Hummer, G. (2004), 'Equation-free: the computer-assisted analysis of complex, multiscale systems', *A. I. Ch. E. Journal* **50**, 1346–1354.
- Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, P. G., Runborg, O. & Theodoropoulos, K. (2003), 'Equation-free, coarse-grained multiscale computation: enabling microscopic simulators to perform system level tasks', *Comm. Math. Sciences* **1**, 715–762.
- Kevrekidis, I. G. & Samaey, G. (2009), 'Equation-free multiscale computation: Algorithms and applications', *Annu. Rev. Phys. Chem.* **60**, 321–44.
- Leitenmaier, L. & Runborg, O. (2021), Heterogeneous multiscale methods for the landau-lifshitz equation, Technical report, <http://arxiv.org/abs/2108.09463>.
- Liu, P., Samaey, G., Gear, C. W. & Kevrekidis, I. G. (2015), 'On the acceleration of spatially distributed agent-based computations: A patch dynamics scheme', *Applied Numerical Mathematics* **92**, 54–69.  
<http://www.sciencedirect.com/science/article/pii/S0168927414002086>
- MacKenzie, T. & Roberts, A. J. (2003), Holistic discretisation of shear dispersion in a two-dimensional channel, in K. Burrage & R. B. Sidje, eds, 'Proc. of 10th Computational Techniques and Applications Conference CTAC-2001', Vol. 44, pp. C512–C530.

- Maclean, J., Bunder, J. E. & Roberts, A. J. (2020), ‘A toolbox of equation-free functions in matlab/octave for efficient system level simulation’, *Numerical Algorithms* .
- Maclean, J. & Gottwald, G. A. (2015), ‘On convergence of higher order schemes for the projective integration method for stiff ordinary differential equations’, *Journal of Computational and Applied Mathematics* **288**, 44–69. <http://www.sciencedirect.com/science/article/pii/S0377042715002149>
- Maier, B. & Verfürth, B. (2021), Numerical upscaling for wave equations with time-dependent multiscale coefficients, Technical report, <http://arxiv.org/abs/2107.14069>.
- Marschler, C., Sieber, J., Berkemer, R., Kawamoto, A. & Starke, J. (2014), ‘Implicit methods for equation-free analysis: Convergence results and analysis of emergent waves in microscopic traffic models’, *SIAM J. Appl. Dyn. Syst.* **13**(2), 1202–1238.
- Oron, A., Davis, S. H. & Bankoff, S. G. (1997), ‘Long-scale evolution of thin liquid films’, *Rev. Mod. Phys.* **69**, 931–980. <http://link.aps.org/abstract/RMP/v69/p931>.
- Petersik, P. (2019–), Equation-free modeling, Technical report, [<https://github.com/pjpetersik/eqnfree>].
- 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](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.