Equation-Free function toolbox for Matlab/Octave: Summary User Manual

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

August 17, 2020

[*] School of Mathematical Sciences, University of Adelaide, South Australia. http://www.maths.adelaide.edu.au/anthony.roberts, http://orcid.org/0000-0001-8930-1552

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

[‡] School of Mathematical Sciences, University of Adelaide, South Australia. mailto:judith.bunder@adelaide.edu.au, http://orcid.org/0000-0001-5355-2288

Abstract

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

Contents

1 Introduction

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

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

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

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

- In 1D space adapt the code at the beginning of `configPatches1.m` for Burgers' PDE (Section 3.2.2).
- In 2D space adapt the code at the beginning of `configPatches2.m` for nonlinear diffusion (Section 3.7.2).
- In 3D space adapt the code at the beginning of `configPatches3.m` for wave propagation through a heterogeneous medium (Section 3.10.2), or the patches of the 3D heterogeneous diffusion of `homoDiffEdgy3.m` (Section 3.13).
- Other provided examples include cases of macroscale *computational homogenisation* of microscale heterogeneity.

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

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

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

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

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

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

2 Projective integration of deterministic ODEs

Chapter contents

2.1 Introduction

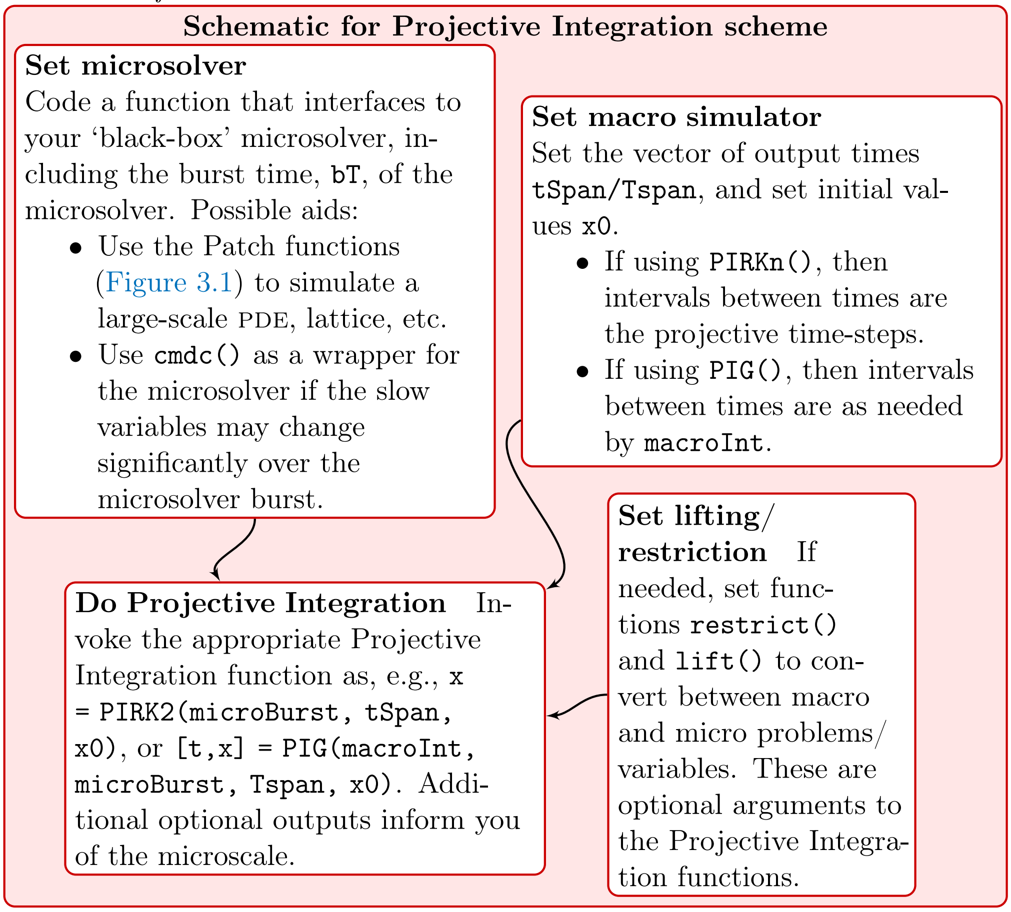
This section provides some good projective integration functions (Gear & Kevrekidis 2003*b*,*c*, Givon et al. 2006, Marschler et al. 2014, Maclean & Gottwald 2015, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive multiscale dynamic simulations/integrations to efficiently compute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

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

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

The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard

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.



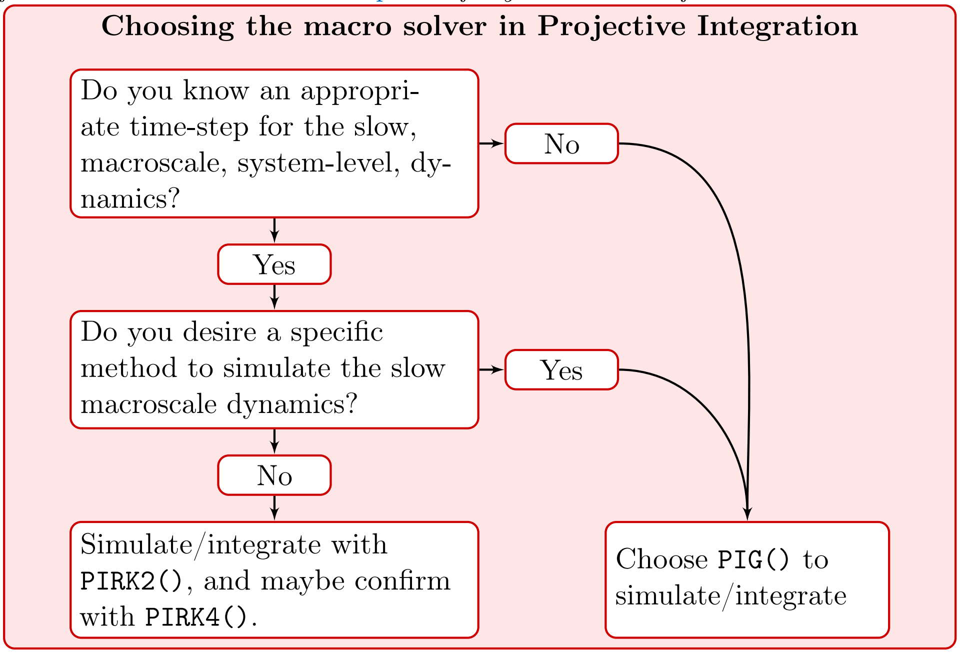
numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

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

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.



method is a standard Matlab/Octave algorithm, or one supplied by the user. `PIG()` should only be used directly in very stiff systems, less stiff systems additionally require `cdmc()`.

- *Constraint-defined manifold computing*, `cdmc()`, is a helper function, based on the method introduced in Gear et al. (2005*a*), that iteratively applies the microsolver and backward projection in time. The result is to project the fast variables close to the slow manifold, without advancing the current time by the burst time of the microsolver. This function reduces errors related to the simulation length of the microsolver in the `PIG` function. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

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()`.

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

Section contents

2.2.1 Introduction

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

```
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.

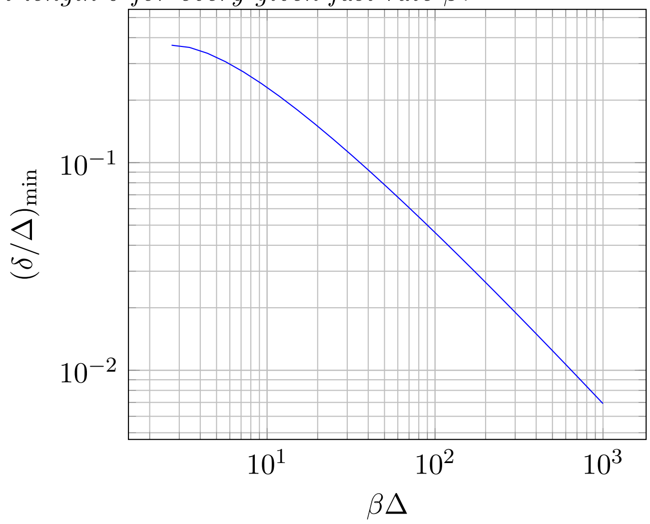
- `tSpan` is an ℓ-vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not use adaptive time-stepping; the macroscale time-steps are (nearly) the steps between elements of `tSpan`.
- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: such `Nans` are carried in the simulation through to the output, and often represent boundaries/edges in spatial fields.
- `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.

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

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

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

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



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

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

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

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

- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding time `tms` and helps visualise details of the solution.

- `rm`, optional, a struct containing the 'remaining' applications of the microBurst required by the Projective Integration method during the calculation of the macrostep:

  - `rm.t` is a column vector of microscale times; and
  - `rm.x` is the array of corresponding burst states.

  The states `rm.x` do not have the same physical interpretation as those in `xms`; the `rm.x` are required in order to estimate the slow vector field during the calculation of the Runge–Kutta increments, and do *not* accurately approximate the macroscale dynamics.

- `svf`, optional, a struct containing the Projective Integration estimates of the slow vector field.

- `svf.t` is a 2ℓ dimensional column vector containing all times at which the Projective Integration scheme is extrapolated along microBurst data to form a macrostep.
- `svf.dx` is a $2\ell \times n$ array containing the estimated slow vector field.

2.2.2 If no arguments, then execute an example

```
if nargin==0
```

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

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

(encoded in function `MMburst()` in the next paragraph). With initial conditions $x(0) = 1$ and $y(0) = 0$, the following code computes and plots a solution over time $0 \leq t \leq 6$ 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 $\Delta = 1$.

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

Upon finishing execution of the example, exit this function.

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

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

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

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

2.3 `egPIMM`: Example projective integration of Michaelis–Menton kinetics

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

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

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

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

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

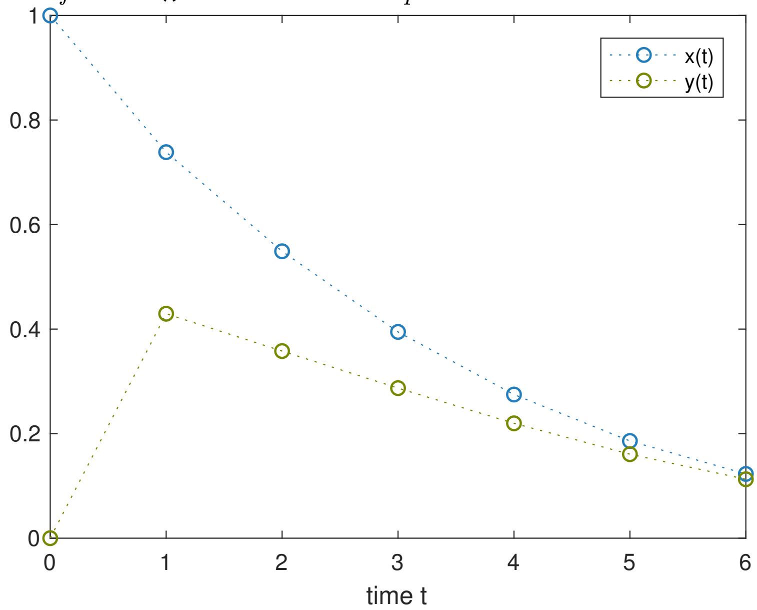
First, the end of this section encodes the computation of bursts of the Michaelis–Menten system in a function `MMburst()`. Second, here set macroscale times of computation and interest into vector `ts`. Then, invoke Projective Integration with `PIRK2()` applied to the burst function, say using bursts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

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

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

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

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




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

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

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

```
ts = 0:-1:-5
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, 0.2*[1;1], 3*MMepsilon);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
```

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

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.*

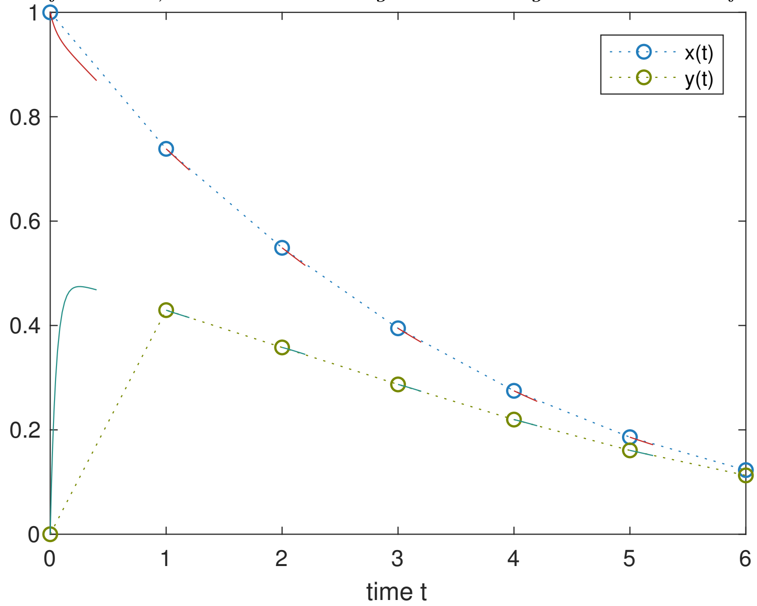
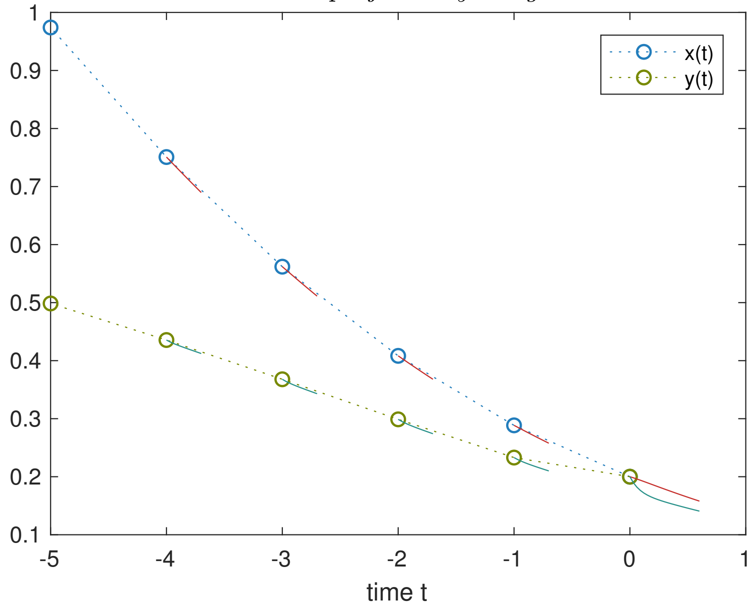


Figure 2.6: *Michaelis–Menten enzyme kinetics at $\epsilon = 0.1$ simulated backward with the projective integration of* `PIRK2()`*: the microscale bursts show the short forward simulations used to projectively integrate backward in time.*



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

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

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

Section contents

2.4.1 Introduction

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

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

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

Inputs:

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

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

where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.2 If no arguments, then execute an example

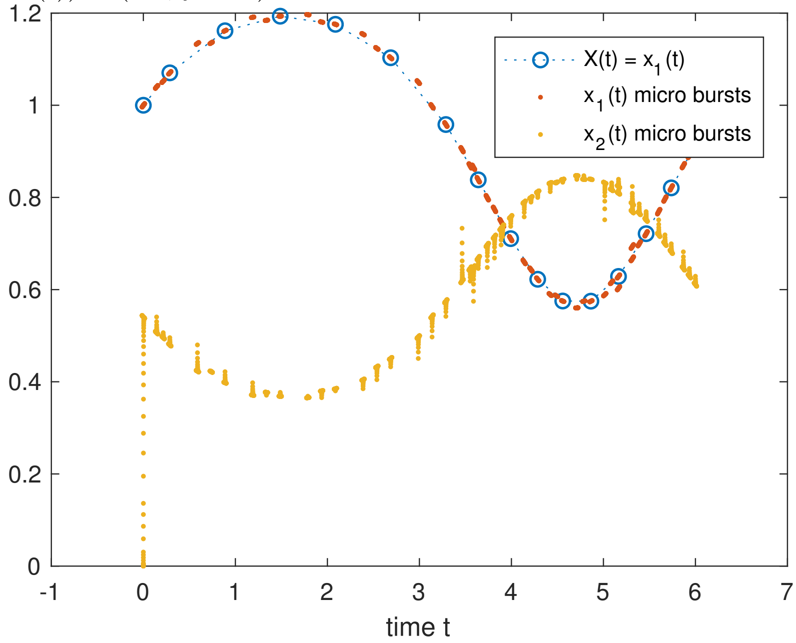
```
if nargin==0
```

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

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

The macroscale variable is $X(t) = x_1(t)$, and the evolution dX/dt is unclear. With initial condition $X(0) = 1$, the following code computes and

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



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.

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

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

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

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

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

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

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

Plot output of this projective integration.

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

Upon finishing execution of the example, exit this function.

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

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

Section contents

2.5.1 Introduction

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

```
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

```
if nargin==0
```

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

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

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

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

Upon finishing execution of the example, exit this function.

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

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

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

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

2.6 `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.

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

Input

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

Output

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

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

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

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

3 Patch scheme for given microscale discrete space system

Chapter contents

3.1 Introduction

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

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

Quick start See Sections 3.2.2 and 3.7.2 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.

3.2 configPatches1(): configures spatial patches in 1D

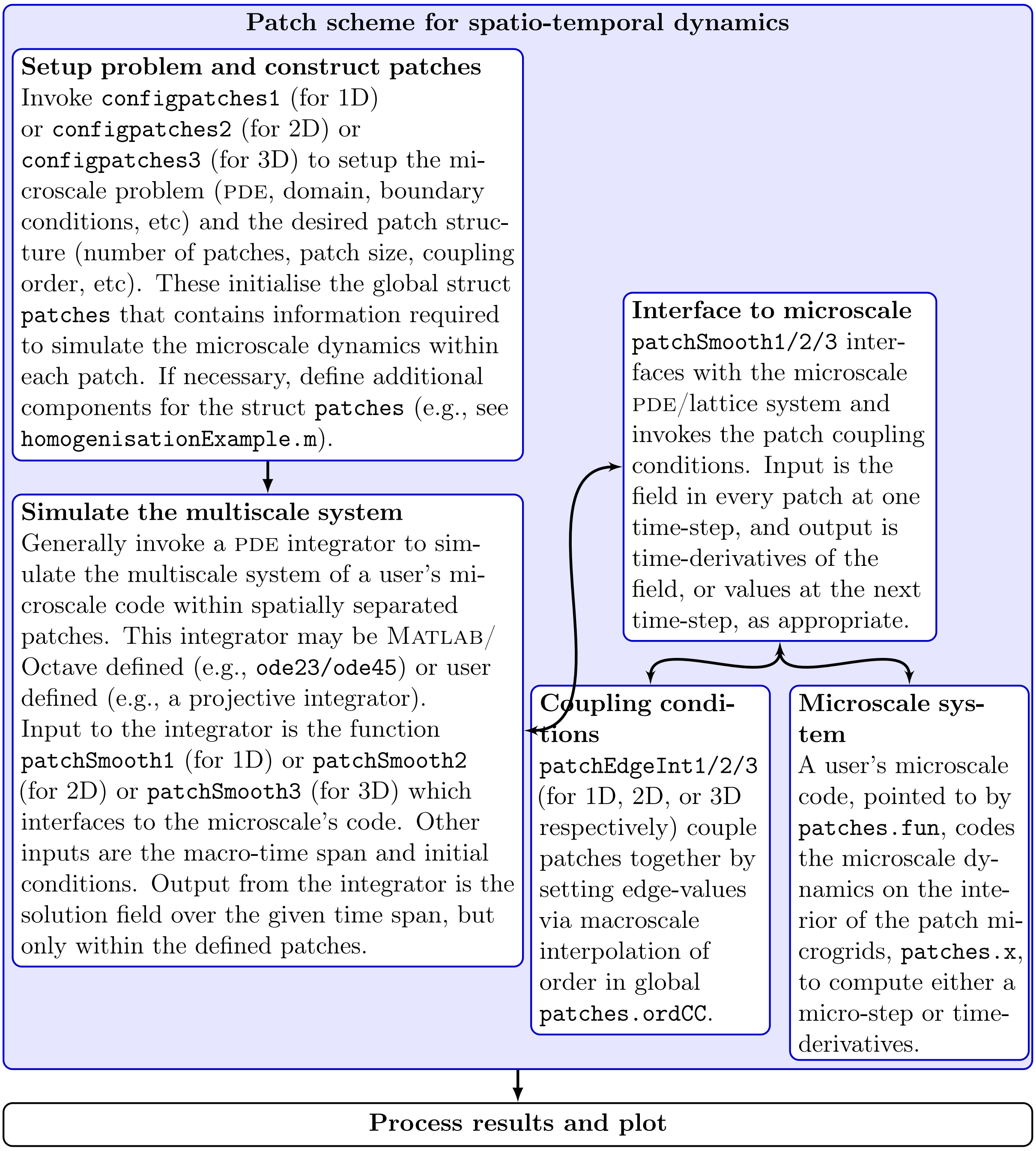
Section contents

3.2.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth1()`. Section 3.2.2 lists an example of its use.

```
function configPatches1(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP ...
                        ,varargin)
global patches
```

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



Input If invoked with no input arguments, then executes an example of simulating Burgers' PDE—see Section 3.2.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space spatial domain of the computation: patches are equi-spaced over the interior of the interval [`Xlim(1)`, `Xlim(2)`].
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the spatial domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC`, must be ≥ -1, is the 'order' of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or -1 gives spectral interpolation; and `ordCC` being odd is for staggered spatial grids.
- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. 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 must be odd so that there is a central lattice point.
- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `'EdgyInt'`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values.
- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `'hetCoeffs'`, *optional*, default empty. Supply an 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 element.
  - 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??

Output The *global* struct `patches` is created and set with the following components.

- `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.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` are the `ordCC`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified.
- `.x` (4D) is `nSubP`$\times 1\times 1\times$`nPatch` array of the regular spatial locations x_{ij} of the ith microscale grid point in the jth 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
  - `[]` 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.

3.2.2 If no arguments, then execute an example

```
if nargin==0
```

The code here shows one way to get started: a user's script may have the following three steps (left-right arrows denote function recursion).

1. configPatches1
2. ode15s integrator $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ user's PDE
3. process results

Establish global patch data struct to point to and interface with a function coding Burgers' PDE: to be solved on 2π-periodic domain, with eight patches, spectral interpolation couples the patches, each patch of half-size ratio 0.2, and with seven microscale points forming each patch.

```
configPatches1(@BurgersPDE,[0 2*pi], nan, 8, 0, 0.2, 7);
```

Set an initial condition, with some microscale randomness.

```
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 `patchsmooth1()` (Section 3.3).

```
if ~exist('OCTAVE_VERSION','builtin')
[ts,us] = ode15s( @patchSmooth1,[0 0.5],u0(:));
else % octave version
[ts,us] = odeOcts(@patchSmooth1,[0 0.5],u0(:));
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(1),clf
if 1, patches.x([1 end],:,:,:)=nan;  us=us.';
else us=reshape(patchEdgyInt1(us.'),[],length(ts));
end
surf(ts,patches.x(:),us)
view(60,40), colormap(hsv)
title('Burgers PDE: patches in space, continuous time')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
ifOurCf2eps(mfilename)
```

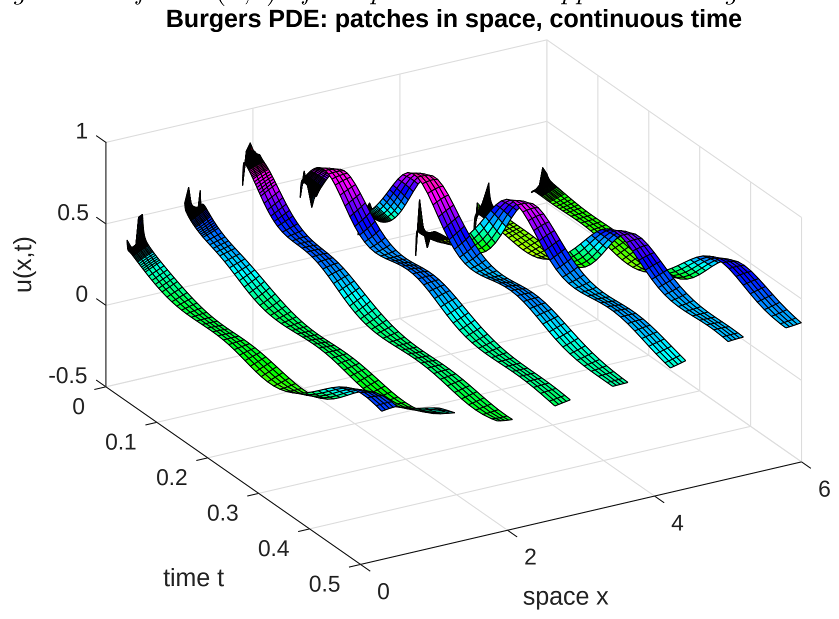
Upon finishing execution of the example, exit this function.

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

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 patchSmooth1()).

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

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




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

3.3 `patchSmooth1()`: 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 `ode15s` or `PIRK2`. This function provides an interface. It mostly assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. However, we have found that microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables to this function using the previously established global struct `patches` (Section 3.2).

```
function dudt=patchSmooth1(t,u)
global patches
```

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `t` is the current time to be passed to the user's time derivative function.

- `patches` a struct set by `configPatches1()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,x)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP` × `nPatch` × `nVars`. Time derivatives must be computed into the same sized array, although herein the patch edge values are overwritten by zeros.
  - `.x` is `nSubP` × `nPatch` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

- `dudt` is `nSubP`·`nPatch`·`nVars` vector of time derivatives, but with patch edge values set to zero.

3.4 `patchEdgeInt1()`: sets edge values from interpolation over the macroscale

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value, or the patch-core average, or the opposite next-to-edge values (this last choice often maintains symmetry). This function is primarily used by `patchSmooth1()` but is also useful for user graphics. A spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). When using core averages (not yet implemented in this version??), assumes they are in some sense *smooth* so that these averages are sensible macroscale variables: then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). Communicates patch-design variables via the global struct `patches`.

```
function u=patchEdgeInt1(u)
global patches
```

Input

- `u` is a vector (or indeed any dim array) of length `nSubP`·`nVars`·`nEnsem`·`nPatch` where there are `nVars`·`nEnsem` field values at each of the points in the `nSubP` × `nPatch` grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is `nSubP` × 1 × 1 × `nPatch` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation integer ≥ -1.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.

- `.Cwtsr` and `.Cwtsl` define the coupling coefficients for finite width interpolation.
- `.EdgyInt` true/false is true for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).
- `.nEnsem` the number of realisations in the ensemble.

Output

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

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

Section contents

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

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

1. configPatches1
2. ode15s $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ heteroDiff
3. process results

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

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

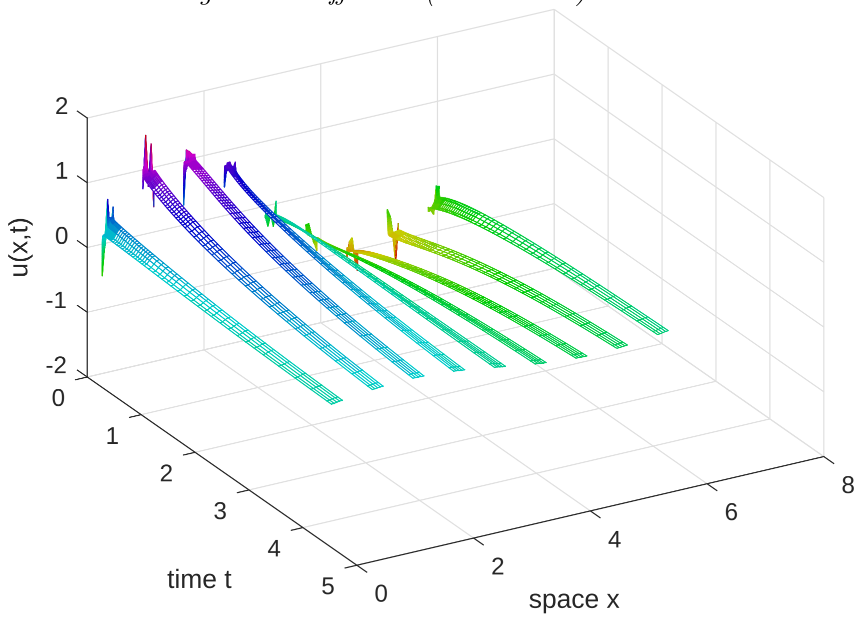
In this 1D space, the macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients.

3.5.1 Script to simulate via stiff or projective integration

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

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

Figure 3.3: the diffusing field $u(x,t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion (Section 3.5).



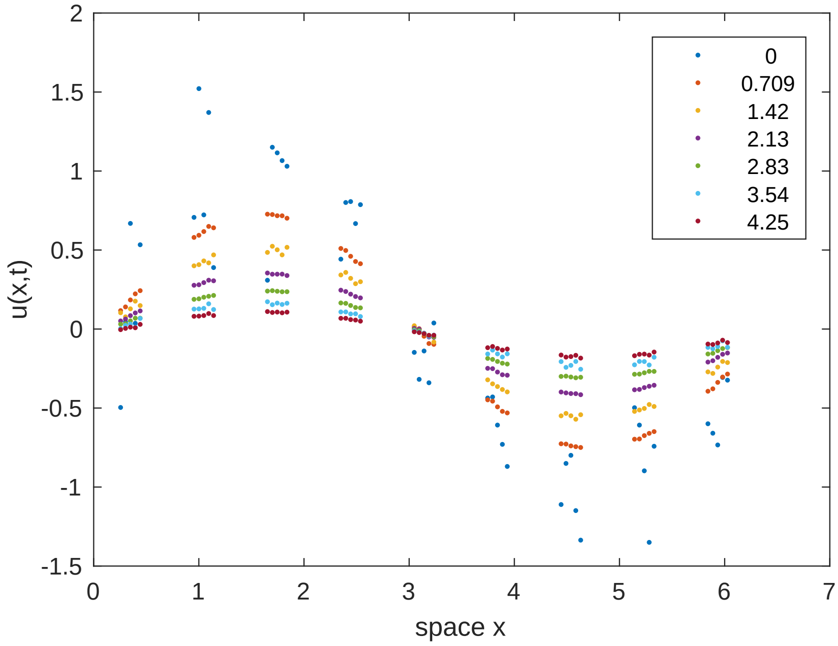
Establish global data struct `patches` for heterogeneous diffusion on 2π-periodic domain. Use nine patches, each patch of half-size ratio 0.2. Quartic (fourth-order) interpolation `ordCC` $= 4$ provides values for the inter-patch coupling conditions. Here include the diffusivity coefficients, repeated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

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

For comparison: conventional integration in time Set an initial condition, and here integrate forward in time using a standard method for stiff systems—because of the simplicity of linear problems this method works quite efficiently here. Integrate the interface `patchSmooth1` (Section 3.3) to the microscale differential equations.

```
u0 = sin(patches.x)+0.3*randn(nSubP,1,1,nPatch);
if ~exist('OCTAVE_VERSION','builtin')
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
else % octave version
[ts,ucts] = odeOcts(@patchSmooth1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

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.



Plot the simulation in Figure 3.3.

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

The code may invoke this integration interface.

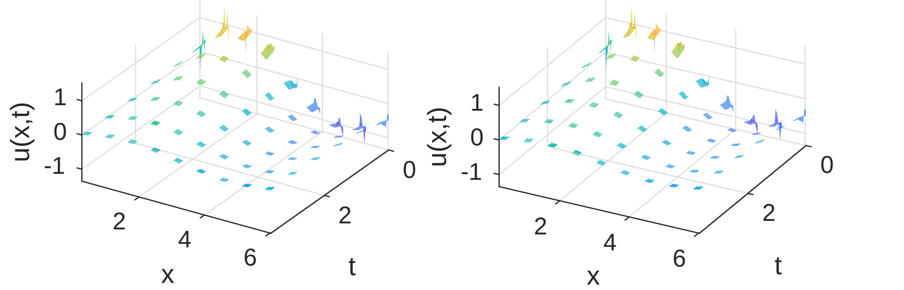
```
function [ts,xs] = odeOcts(dxdt,tSpan,x0)
    if length(tSpan)>2, ts = tSpan;
    else ts = linspace(tSpan(1),tSpan(end),21)';
    end
    lsode_options('integration method','non-stiff');
    xs = lsode(@(x,t) dxdt(t,x),x0,ts);
end
```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), 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)

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.



2. PIRK2 ↔ heteroBurst ↔ micro-integrator ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

```
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.

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

Plot the macroscale predictions to draw Figure 3.4.

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

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

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

End of this example script.

3.5.2 `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 `patchSmooth1`, Section 3.3), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector, or possibly array of phase-shifts, of diffusivities c_i have previously been stored in struct `patches.cs`.

```
function ut = heteroDiff(t,u,x)
  global patches
  dx = diff(x(2:3));   % space step
  i = 2:size(u,1)-1;   % interior points in a patch
  ut = nan(size(u)); % preallocate output array
  ut(i,:,:,:) = diff(patches.cs.*diff(u))/dx^2;
  % if set, include heterogeneous Burgers' advection
  if isfield(patches,'b') % check for advection coeffs
      buu = patches.b.*u.^2;
      ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
  end
end% function
```

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

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSmooth1`. Try `ode23` or `rk2Int`, although `ode45` may give smoother results.

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
        if ~exist('OCTAVE_VERSION','builtin')
        [ts,ucts] = ode23( @patchSmooth1,[ti ti+bT],ui(:));
        else % octave version
        [ts,ucts] = rk2Int(@patchSmooth1,[ti ti+bT],ui(:));
        end
end
```

Fin.

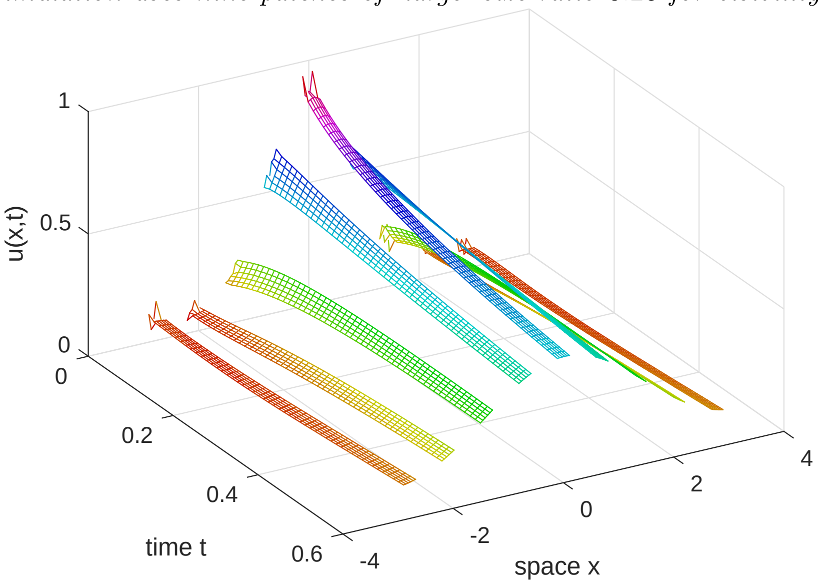
3.6 `homoDiffEdgy1`: computational homogenisation of a 1D diffusion by simulation on small patches

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

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

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

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.*



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

3.6.1 Script code to simulate heterogeneous diffusion systems

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

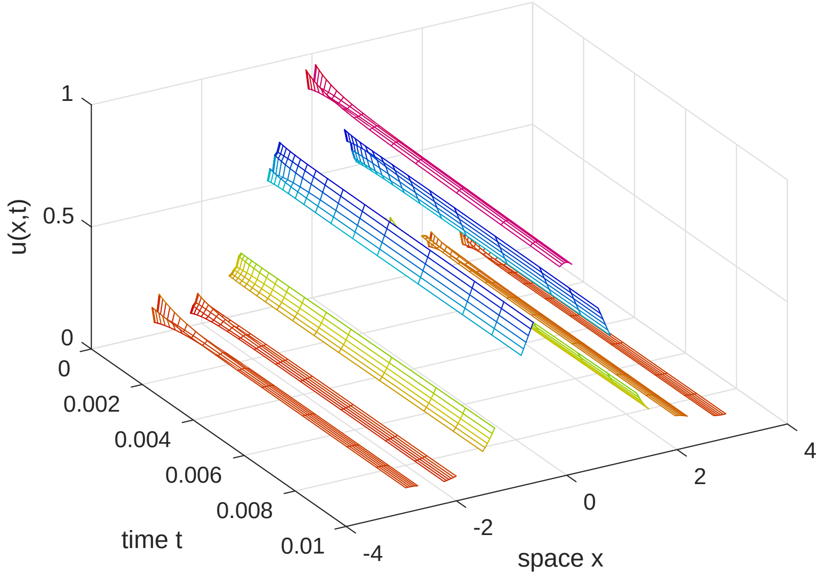
1. configPatches1, and add micro-information
2. ode15s $\leftrightarrow$ patchSmooth1 $\leftrightarrow$ heteroDiff
3. plot the simulation
4. use patchSmooth1 to explore the Jacobian

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

```
mPeriod = 3%randi([2 5])
% set random diffusion coefficients
cHetr=exp(0*randn(mPeriod,1));
%cHetr = [3.966;2.531;0.838;0.331;7.276];
cHetr = cHetr*mean(1./cHetr) % normalise
```

Establish the global data struct `patches` for the microscale heterogeneous

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.



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

```
global patches
nPatch = 9
ratio = 0.25;
nSubP = mPeriod+3 %randi([mPeriod+1 2*mPeriod+2])
nEnsem = mPeriod % number realisations in ensemble
if mod(nSubP,mPeriod)==2, nEnsem=1, end
configPatches1(@heteroDiff,[-pi pi],nan,nPatch ...
    ,4,ratio,nSubP,'EdgyInt',true,'nEnsem',nEnsem ...
    ,'hetCoeffs',cHetr);
```

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

```
u0 = 0.8*exp(-patches.x.^2)+0.2*rand(nSubP,1,nEnsem,nPatch);
du0dt = patchSmooth1(0,u0(:));
```

Integrate using standard integrators.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth1, [0 0.6], u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth1, 0.6*linspace(0,1).^2, u0(:));
end
```

Plot space-time surface of the simulation We want to see the edge values of the patches, so we adjoin a row of 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.

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

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

```
for p=1:2
  switch p
  case 1, j=find(ts<0.01);
  case 2, [~,j]=min(abs(ts(:)-linspace(ts(1),ts(end),50)));
  end
  figure(p),clf
  mesh(ts(j),xs(:),us(:,j))
  view(60,40), colormap(0.8*hsv)
  xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
  ifOurCf2eps([mfilename 'U' num2str(p)])
end
pause(3)
```

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

```
nPatch = 13
ratio = 0.01;

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

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

```
  u0 = zeros(nSubP,1,nEnsem,nPatch);
  u0([1 end],:,:,:)=nan; u0=u0(:);
  i=find(~isnan(u0));
  nJ=length(i);
  Jac=nan(nJ);
  for j=1:nJ
    u0(i)=((1:nJ)==j);
    dudt=patchSmooth1(0,u0);
    Jac(:,j)=dudt(i);
  end
  nonSymmetric=norm(Jac-Jac')
  assert(nonSymmetric<5e-9,'failed symmetry')
  Jac(abs(Jac)<1e-12)=0;
```

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

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

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

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

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

End of the main script.

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
```

3.6.2 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 patchSmooth1, Section 3.3), computes the time derivative (3.1) at each point in the interior of a patch, output in ut. The column vector, or possibly array of phase-shifts, of diffusivities c_i have previously been stored in struct patches.cs.

```
function ut = heteroDiff(t,u,x)
  global patches
  dx = diff(x(2:3));    % space step
  i = 2:size(u,1)-1;    % interior points in a patch
  ut = nan(size(u)); % preallocate output array
  ut(i,:,:,:) = diff(patches.cs.*diff(u))/dx^2;
  % if set, include heterogeneous Burgers' advection
  if isfield(patches,'b') % check for advection coeffs
      buu = patches.b.*u.^2;
      ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
  end
end% function
```

Fin.

3.7 `configPatches2()`: configures spatial patches in 2D

Section contents

3.7.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSmooth2()`. Section 3.7.2 lists an example of its use.

```
function configPatches2(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP ...
    ,varargin)
global patches
```

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 3.7.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x,y)`, that computes time-derivatives (or time-steps) of quantities on the 2D micro-grid within all the 2D patches.

- `Xlim` array/vector giving the macro-space domain of the computation: patches are distributed equi-spaced over the interior of the rectangle [`Xlim(1)`, `Xlim(2)`] $\times$ [`Xlim(3)`, `Xlim(4)`]: if `Xlim` is of length two, then the domain is the square domain of the same interval in both directions.

- `BCs` eventually and somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.

- `nPatch` determines the number of equi-spaced spaced patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number 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 edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.

- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. 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. If scalar, then use the same ratio in both directions, otherwise `ratio(1:2)` gives the ratio in each direction.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise `nSubP(1:2)` gives the number in each direction. If not using `EdgyInt`, then must be odd so that there is a central micro-grid point in each patch.

- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for 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.
- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `'hetCoeffs'`, *optional*, default empty. Supply an 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. 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 $(1, 1)$-element.
  - 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.

Output The *global* struct `patches` is created and set with the following components.

- `.fun` is the name of the user's function `fun(t,u,x,y)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC`$\times$2-array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified.
- `.x` (6D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1$ array of the regular spatial locations x_{ij} of the microscale grid points in every patch.

- `.y` (6D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the regular spatial locations y_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times$ (`nSubP(2)` $- 1) \times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` > 1, (`nSubP(1)` $- 1) \times$ (`nSubP(2)` $- 1) \times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.

3.7.2 If no arguments, then execute an example

```
if nargin==0
```

The code here shows one way to get started: a user's script may have the following three steps (arrows indicate function recursion).

1. configPatches2
2. ode23 integrator ↔ patchSmooth2 ↔ user's PDE
3. process results

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.4 (relatively large for visualisation), and with 5×5 points forming each patch. Roberts et al. (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
nSubP = 5;
configPatches2(@nonDiffPDE,[-3 3 -2 2], nan, [9 7] ...
    , 0, 0.4, nSubP ,'EdgyInt',false);
```

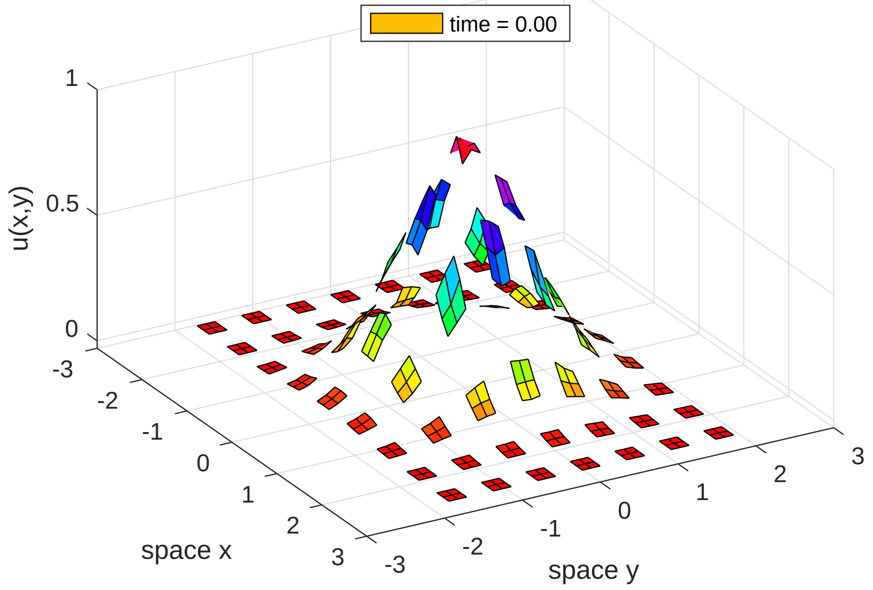
Set a perturbed-Gaussian initial condition using auto-replication of the spatial grid.

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

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

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

Figure 3.8: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 3.9 plots the computed field at time $t = 3$.



Start by showing the initial conditions of Figure 3.8 while the simulation computes.

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

Optionally save the initial condition to graphic file for Figure 3.8.

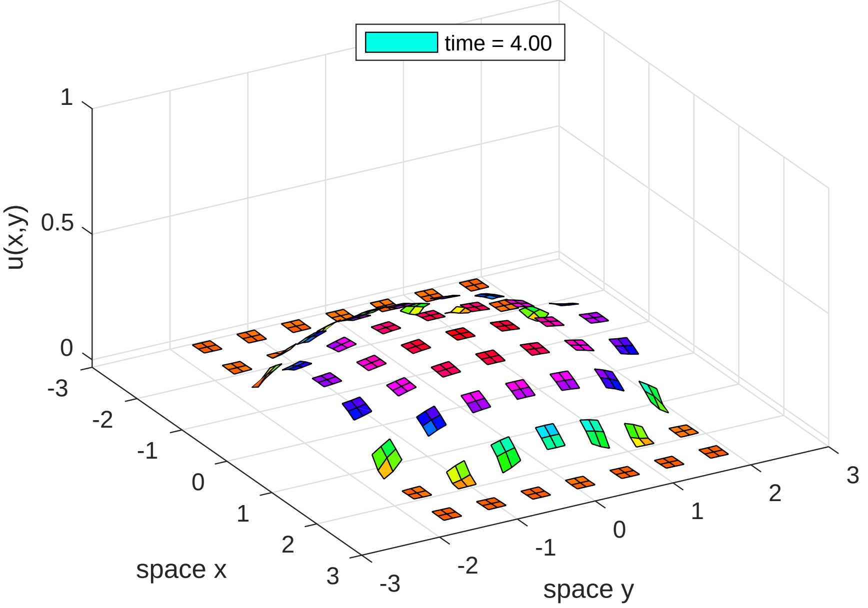
```
ifOurCf2eps([mfilename 'ic'])
```

Integrate in time using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker. Ask for output at non-uniform times as the diffusion slows.

```
disp('Wait to simulate nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
drawnow
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth2,linspace(0,2).^2,u0(:));
else % octave version is quite slow for me
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSmooth2,[0 1],u0(:));
end
```

Animate the computed simulation to end with Figure 3.9. Use `patchEdgeInt2` to interpolate patch-edge values (even if not drawn).

Figure 3.9: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE *with initial condition in Figure 3.8.*




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

Upon finishing execution of the example, exit this function.

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

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

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

3.8 `patchSmooth2()`: 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 `ode15s` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables to this function via the previously established global struct `patches`.

```
function dudt = patchSmooth2(t,u)
global patches
```

Input

- `u` is a vector of length `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` where there are `nVars` $\cdot$ `nEnsem` field values at each of the points in the `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nPatch(1)` $\times$ `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,x,y)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nVars` $\times$ `nEsem` $\times$ `nPatch(1)` $\times$ `nPatch(2)`. Time derivatives must be computed into the same sized array, but herein the patch edge-values are overwritten by zeros.
  - `.x` is `nSubP(1)` $\times 1 \times 1 \times 1 \times$ 1`nPatch(1)` $\times 1$ array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

- `dudt` is `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` vector of time derivatives, but with patch edge-values set to zero.

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

Couples 2D patches across 2D space by computing their edge values via macroscale interpolation. Assumes that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values?? Communicate patch-design variables via the global struct `patches`.

```
function u = patchEdgeInt2(u)
global patches
```

Input

- u is a vector of length nSubP1 · nSubP2 · nVars · nEnsem · nPatch1 · nPatch2 where there are nVars field values at each of the points in the nSubP1 · nSubP2 · nPatch1 · nPatch2 grid on the nPatch1 · nPatch2 array of patches.
- patches a struct set by configPatches2() which includes the following information.
  - .x is nSubP1 $\times 1 \times 1 \times 1 \times$ nPatch1 $\times 1$ array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - .y is similarly $\times 1$nSubP1 $\times 1 \times 1 \times 1 \times$ nPatch1 array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - .ordCC is order of interpolation, currently (July 2020) only $\{0, 2, 4, \ldots\}$
  - .stag in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - .Cwtsr and .Cwtsl define the coupling coefficients for finite width interpolation.
  - .EdgyInt true/false is true for interpolating patch-edge values from opposite next-to-edge values (often preserves symmetry).

Output

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

3.10 configPatches3(): configures spatial patches in 3D

Section contents

3.10.1 Introduction

Makes the struct patches for use by the patch/gap-tooth time derivative/step function patchSmooth3(). Sections 3.10.2 and 3.13 lists an example of its use.

```
function configPatches3(fun,Xlim,BCs,nPatch,ordCC,ratio,nSubP ...
    ,varargin)
global patches
```

Input If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE.

- `fun` is the name of the user function, `fun(t,u,x,y,z)`, that computes time-derivatives (or time-steps) of quantities on the 3D micro-grid within all the 3D patches.
- `Xlim` array/vector giving the macro-space domain of the computation: patches are distributed equi-spaced over the interior of the rectangle [`Xlim(1)`, `Xlim(2)`] $\times$ [`Xlim(3)`, `Xlim(4)` $\times$ [`Xlim(5)`, `Xlim(6)`]: if `Xlim` is of length two, then the domain is the cubic domain of the same interval in all three directions.
- `BCs` eventually and somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` determines the number of equi-spaced spaced patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:3)` gives the number 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 edge-values of the patches: currently must be $0, 2, 4, \ldots$; where 0 gives spectral interpolation.
- `ratio` (real) is the ratio of (depending upon `EdgyInt`) either the half-width or full-width of a patch to the spacing of the patch mid-points. 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. If scalar, then use the same ratio in both directions, otherwise `ratio(1:3)` gives the ratio in each direction.
- `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(1:3)` gives the number in each direction. If not using `EdgyInt`, then must be odd so that there is a central micro-grid point/planes in each patch.
- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for 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.

- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.
- `'hetCoeffs'`, *optional*, default empty. Supply a 3/4D 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. 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 $(1, 1)$-element.
  - 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.

Output The *global* struct `patches` is created and set with the following components.

- `.fun` is the name of the user's function `fun(t,u,x,y,z)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.stag` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling—not yet implemented.
- `.Cwtsr` and `.Cwtsl` are the `ordCC`$\times 3$-array of weights for the inter-patch interpolation onto the right/top and left/bottom edges (respectively) with patch:macroscale ratio as specified.
- `.x` (8D) is `nSubP(1)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(1)` $\times 1 \times 1$ array of the regular spatial locations x_{ijk} of the microscale grid points in every patch.
- `.y` (8D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the regular spatial locations y_{ijk} of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the regular spatial locations z_{ijk} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

- `.le`, `.ri`, `.bo`, `.to`, `.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) $\times n_c \times m_x m_y$ 5D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.

3.10.2 If no arguments, then execute an example

```
if nargin==0
```

The code here shows one way to get started: a user's script may have the following three steps (arrows indicate function recursion).

1. configPatches3
2. ode23 integrator ↔ patchSmooth3 ↔ 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 decay time scale is roughly one.

```
mPeriod = [2 2 2];
cHetr = exp(0.3*randn([mPeriod 3]));
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 of half-size ratio 0.4 (relatively large for visualisation), and with 4^3 points forming each patch.

```
configPatches3(@heteroWave3,[-pi pi], nan, 5 ...
    , 0, 0.35, mPeriod+2 ,'EdgyInt',true ...
    ,'hetCoeffs',cHetr);
```

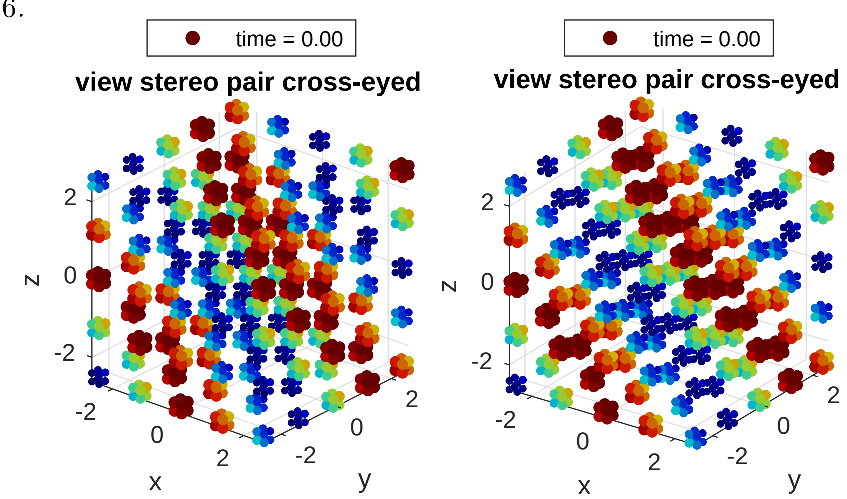
Set a wave initial state using auto-replication of the spatial grid, and as Figure 3.10 shows.

```
u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
v0 =    -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
uv0 = cat(4,u0,v0);
```

Integrate in time using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker.

```
disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth3,linspace(0,6),uv0(:));
```

Figure 3.10: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave PDE*: Figure 3.11 plots the computed field at time $t = 6$.*




```
else %disp('octave version is very slow for me')
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSmooth3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 3.11. Use `patchEdgeInt3` to interpolate patch-edge values (even if not drawn) in order to most easily reconstruct the array data structure.

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

```
figure(1), clf, colormap(0.8*jet)
xs = patches.x+0*patches.y+0*patches.z;
ys = patches.y+0*patches.x+0*patches.z;
zs = patches.z+0*patches.y+0*patches.x;
xs([1 end],:,:,:)=nan;
xs(:,[1 end],:,:)=nan;
xs(:,:,[1 end],:)=nan;
j=find(~isnan(xs));
```

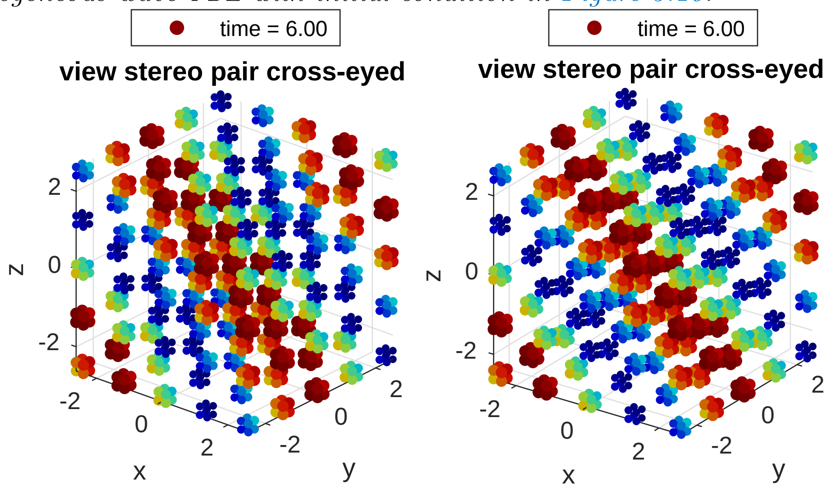
The functions `pix()` and `col()` map the u-data values to the size of the dots and to the colour of the dots, respectively.

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

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
```

Figure 3.11: 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 3.10.




```
  for p=1:2
    subplot(1,2,p)
    if (i==1)| exist('OCTAVE_VERSION','builtin')
      scat(p) = scatter3(xs(j),ys(j),zs(j) ...
              ,pix(u(j)),col(u(j)),'filled');
      axis equal, caxis(col([0 1])), view(45-5*p,25)
      xlabel('x'), ylabel('y'), zlabel('z')
      title('view stereo pair cross-eyed')
    else % in matlab just update values
      set(scat(p),'CData',col(u(j)),'SizeData',pix(u(j)));
    end
    legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
  end
```

Optionally save the initial condition to graphic file for Figure 3.8, and optionally save the last plot.

```
  if i==1,
    ifOurCf2eps([mfilename 'ic'])
    disp('Type space character to animate simulation')
    pause
  else pause(0.05)
  end
end% i-loop over all times
ifOurCf2eps([mfilename 'fin'])
```

Upon finishing execution of the example, exit this function.

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

3.10.3 `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 `patchSmooth3`, Section 3.11), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of heterogeneous coefficients, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

```
function ut = heteroWave3(t,u,x,y,z)
  global patches
  dx = diff(x(2:3));  % x space step
  dy = diff(y(2:3));  % y space step
  dz = diff(z(2:3));  % z space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  iz = 2:size(u,3)-1; % z interior points in a patch
  ut = nan(size(u));  % preallocate output array
  ut(ix,iy,iz,1,:) = u(ix,iy,iz,2,:);
  ut(ix,iy,iz,2,:) ...
  =diff(patches.cs(:,iy,iz,1,:).*diff(u(:,iy,iz,1,:),1),1)/dx^2 ...
  +diff(patches.cs(ix,:,iz,2,:).*diff(u(ix,:,iz,1,:),1,2),1,2)/dy^2 ...
  +diff(patches.cs(ix,iy,:,3,:).*diff(u(ix,iy,:,1,:),1,3),1,3)/dz^2;
end% function
```

3.11 `patchSmooth3()`: 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 `ode15s` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables to this function via the previously established global struct `patches`.

```
function dudt = patchSmooth3(t,u)
global patches
```

Input

- `u` is a vector of length `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` where there are `nVars` $\cdot$ `nEnsem` field values at each of the points in the `nSubP(1)`$\times$`nSubP(2)`$\times$`nSubP(3)`$\times$`nPatch(1)`$\times$`nPatch(2)`$\times$`nPatch(3)` 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,x,y,z)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nSubP(3)` $\times$ `nVars` $\times$ `nEsem` $\times$ `nPatch(1)` $\times$ `nPatch(2)` $\times$ `nPatch(3)`. Time derivatives must be computed into the same sized array, but herein the patch edge-values are overwritten by zeros.
  - `.x` is `nSubP(1)` $\times 1 \times 1 \times 1 \times 1$`nPatch(1)` $\times 1 \times 1$ array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.z` is similarly $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the spatial locations z_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

Output

- `dudt` is `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` vector of time derivatives, but with patch edge-values set to zero.

3.12 `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 that the patch centre-values are sensible macroscale variables, and patch face values are determined by macroscale interpolation of the patch-centre values?? Communicate patch-design variables via the global struct `patches`.

```
function u = patchEdgeInt3(u)
global patches
```

Input

- `u` is a vector of length `nSubP1`$\cdot$`nSubP2`$\cdot$`nSubP3`$\cdot$`nVars`$\cdot$`nEnsem`$\cdot$`nPatch1`$\cdot$ `nPatch2` $\cdot$ `nPatch3` where there are `nVars` field values at each of the points in the `nSubP1`$\cdot$`nSubP2`$\cdot$`nSubP3`$\cdot$`nPatch1`$\cdot$`nPatch2`$\cdot$`nPatch3` grid on the `nPatch1` $\cdot$ `nPatch2` $\cdot$ `nPatch3` array of patches.
- `patches` a struct set by `configPatches3()` which includes the following information.
  - `.x` is `nSubP1` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch1` $\times 1 \times 1$ array of the spatial locations x_{ijk} of the microscale grid points in every patch.

Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.

- `.y` is similarly $\times 1$`nSubP2`$\times 1 \times 1 \times 1 \times 1 \times$`nPatch2`$\times 1$ array of the spatial locations y_{ijk} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
- `.z` is similarly $\times 1 \times 1$`nSubP3`$\times 1 \times 1 \times 1 \times 1 \times$`nPatch3` array of the spatial locations z_{ijk} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
- `.ordCC` is order of interpolation, currently (Aug 2020) only $\{0, 2, 4, \ldots\}$
- `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
- `.Cwtsr` and `.Cwtsl` define the coupling coefficients for finite width interpolation.
- `.EdgyInt` true/false is true for interpolating patch-face values from opposite next-to-face values (often preserves symmetry).

Output

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

3.13 `homoDiffEdgy3`: computational homogenisation of a 3D diffusion via simulation on small patches

Simulate heterogeneous diffusion in 3D space on 3D patches as an example application. Then compute macroscale eigenvalues of the patch scheme applied to this heterogeneous diffusion to validate and to compare various orders of inter-patch interpolation.

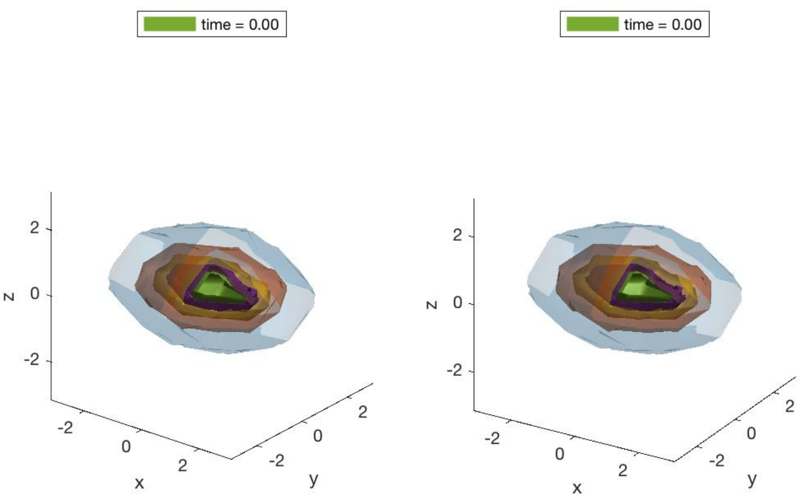
This code extends to 3D the 2D code discussed in **??**. First set random heterogeneous diffusivities of random (small) period in each of the three directions. Crudely normalise by the harmonic mean so the decay time scale is roughly one.

```
mPeriod = randi([2 3],1,3)
cHetr = exp(0.3*randn([mPeriod 3]));
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.

```
nSubP=mPeriod+2;
nPatch=[5 5 5];
configPatches3(@heteroDiff3, [-pi pi], nan, nPatch ...
```

Figure 3.12: initial field $u(x, y, z, 0)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside. Figure 3.13 plots the isosurfaces of the computed field at time $t = 0.3$.*




```
    ,0, 0.3, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr );
```

3.13.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 3.12.

```
global patches
u0 = exp(-patches.x.^2/4-patches.y.^2/2-patches.z.^2);
u0 = u0.*(1+0.3*rand(size(u0)));
```

Integrate using standard integrators, unevenly spaced in time to better display transients.

```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSmooth3, 0.3*linspace(0,1,50).^2, u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSmooth3, 0.3*linspace(0,1).^2, u0(:));
end
```

Plot the solution as an animation over time.

```
figure(1), clf
rgb=get(gca,'defaultAxesColorOrder');
colormap(0.8*hsv)
```

Get spatial coordinates of patch interiors.

```
x = reshape( patches.x([2:end-1],:,:,:) ,[],1);
y = reshape( patches.y(:,[2:end-1],:,:) ,[],1);
z = reshape( patches.z(:,:,[2:end-1],:) ,[],1);
```

For every time step draw the surface and pause for a short display.

```
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.

```
  u = reshape( us(i,:) ,[nSubP nPatch]);
  u = u([2:end-1],[2:end-1],[2:end-1],:,:,:);
  u = reshape( permute(u,[1 4 2 5 3 6]) ...
      , [numel(x) numel(y) numel(z)]);
```

Optionally cut-out the front corner so we can see inside.

```
  u( (x>0) & (y'<0) & (shiftdim(z,-2)>0) ) = nan;
```

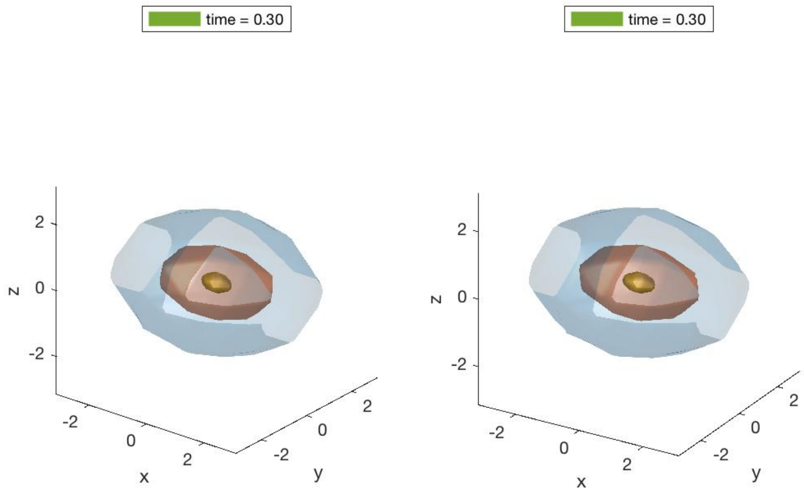
The `isosurface` function requires us to transpose x and y.

```
v = permute(u,[2 1 3]);
```

Draw cross-eyed stereo view of some isosurfaces.

```
  clf;
  for p=1:2
    subplot(1,2,p)
    for iso=5:-1:1
       isov=(iso-0.5)/5;
       hsurf(iso) = patch(isosurface(x,y,z,v,isov));
       isonormals(x,y,z,v,hsurf(iso))
       set(hsurf(iso) ,'FaceColor',rgb(iso,:) ...
           ,'EdgeColor','none' ...
           ,'FaceAlpha',iso/5);
       hold on
    end
    axis equal, view(45-7*p,25)
    axis(pi*[-1 1 -1 1 -1 1])
    xlabel('x'), ylabel('y'), zlabel('z')
    legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
    camlight, lighting gouraud
    hold off
  end% each p
  if i==1 % pause for the viewer
       makeJpeg=false;
       if makeJpeg, print(['Figs/' mfilename 't0'],'-djpeg'), end
       disp('Press any key to start animation')
       pause
```

Figure 3.13: final field $u(x, y, z, 0.3)$ of the patch scheme applied to a heterogeneous diffusion PDE. *Plotted are the isosurfaces at field values $u = 0.1, 0.3, \ldots, 0.9$, with the front quadrant omitted so you can see inside.*




```
    else pause(0.05)
    end
```

Finish the animation loop, and optionally output the isosurfaces of the final field, Figure 3.13.

```
end%for over time
if makeJpeg, print(['Figs/' mfilename 'tFin'],'-djpeg'), end
```

3.13.2 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.

```
ratio = 0.025*(1+rand(1,3))
nSubP=randi([3 5],1,3)
nPatch=[3 3 3]
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.

```
    u0 = zeros([nSubP,1,nEnsem,nPatch]);
    u0([1 end],:,:,:) = nan;
    u0(:,[1 end],:,:) = nan;
```

```
    u0(:,:,[1 end],:) = nan;
    i = find(~isnan(u0));
    sizeJacobian = length(i)
    assert(sizeJacobian<4000 ...
        ,'Jacobian is too big to quickly generate and analyse')
```

Store this many eigenvalues in array across different orders of interpolation.

```
nLeadEvals=prod(nPatch)+max(nPatch);
leadingEvals=[];
```

Evaluate eigenvalues for spectral as the base case for polynomial interpolation of order $2, 4, \ldots$.

```
maxords=6;
for ord=0:2:maxords
    ord=ord
```

Configure with same heterogeneity.

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

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

```
    jac = nan(length(i));
    for j = 1:length(i)
      u = u0(:)+(i(j)==(1:numel(u0))');
      tmp = patchSmooth3(0,u);
      jac(:,j) = tmp(i);
    end
```

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

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

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

```
    [evecs,evals] = eig((jac+jac')/2,'vector');
    biggestImag=max(abs(imag(evals)));
    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.

```
    [~,k] = sort(-real(evals));
    evals=evals(k); evecs=evecs(:,k);
    if ord==0, nZeroEv=sum(abs(evals(:))<1e-5), end
    if ord==0, evec0=evecs(:,1:nZeroEv*nLeadEvals);
```

```
    else % find evec closest to that of each leading spectral
        [~,k]=max(abs(evecs'*evec0));
        evals=evals(k); % re-sort in corresponding order
    end
    leadingEvals=[leadingEvals evals(nZeroEv*(1:nLeadEvals))];
end
disp('     spectral    quadratic      quartic  sixth-order ...')
leadingEvals=leadingEvals
```

3.13.3 `heteroDiff3()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSmooth3`, Section 3.11), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, c^x_{ijk}, c^y_{ijk} and c^z_{ijk}, have previously been stored in `patches.cs` (4D).

```
function ut = heteroDiff3(t,u,x,y,z)
  global patches
  dx = diff(x(2:3));  % x space step
  dy = diff(y(2:3));  % y space step
  dz = diff(z(2:3));  % z space step
  ix = 2:size(u,1)-1; % x interior points in a patch
  iy = 2:size(u,2)-1; % y interior points in a patch
  iz = 2:size(u,3)-1; % y interior points in a patch
  ut = nan(size(u));  % preallocate output array
  ut(ix,iy,iz,:,:,:,:,:) ...
  = diff(patches.cs(:,iy,iz,1,:).*diff(u(:,iy,iz,:,:,:,:,:),1),1)/dx^2 ...
   +diff(patches.cs(ix,:,iz,2,:).*diff(u(ix,:,iz,:,:,:,:,:),1,2),1,2)/dy^2 ..
   +diff(patches.cs(ix,iy,:,3,:).*diff(u(ix,iy,:,:,:,:,:,:),1,3),1,3)/dz^2;
end% function
```

Fin.

Bibliography

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. & Kevrekidis, I. G. (2017), 'Good coupling for the multiscale patch scheme on systems with microscale heterogeneity', *J. Computational Physics* **337**, 154–174.

Cao, M. & Roberts, A. J. (2016), 'Multiscale modelling couples patches of nonlinear wave-like simulations', *IMA J. Applied Maths.* **81**(2), 228–254.

Frewen, T. A., Hummer, G. & Kevrekidis, I. G. (2009), 'Exploration of effective potential landscapes using coarse reverse integration', *The Journal of Chemical Physics* **131**(13), 134104.

Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005*a*), 'Projecting to a slow manifold: singularly perturbed systems and legacy codes', *SIAM J. Applied Dynamical Systems* **4**(3), 711–732.
http://www.siam.org/journals/siads/4-3/60829.html

Gear, C. W., Kaper, T. J., Kevrekidis, I. G. & Zagaris, A. (2005*b*), 'Projecting to a slow manifold: Singularly perturbed systems and legacy codes', *SIAM Journal on Applied Dynamical Systems* **4**(3), 711–732.

Gear, C. W. & Kevrekidis, I. G. (2003*a*), 'Computing in the past with forward integration', *Phys. Lett. A* **321**, 335–343.

Gear, C. W. & Kevrekidis, I. G. (2003*b*), 'Projective methods for stiff differential equations: Problems with gaps in their eigenvalue spectrum', *SIAM Journal on Scientific Computing* **24**(4), 1091–1106.
http://link.aip.org/link/?SCE/24/1091/1

Gear, C. W. & Kevrekidis, I. G. (2003*c*), 'Telescopic projective methods for parabolic differential equations', *Journal of Computational Physics* **187**, 95–109.

Givon, D., Kevrekidis, I. G. & Kupferman, R. (2006), 'Strong convergence of projective integration schemes for singularly perturbed stochastic differential systems', *Comm. Math. Sci.* **4**(4), 707–729.

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.

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

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

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.

Petersik, P. (2019–), Equation-free modeling, Technical report, [https://github.com/pjpetersik/eqnfree].

Roberts, A. J. & Kevrekidis, I. G. (2007), 'General tooth boundary conditions for equation free modelling', *SIAM J. Scientific Computing* **29**(4), 1495–1510.

Roberts, A. J., MacKenzie, T. & Bunder, J. (2014), 'A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions', *J. Engineering Mathematics* **86**(1), 175–207.
http://arxiv.org/abs/1103.1187

Samaey, G., Kevrekidis, I. G. & Roose, D. (2005), 'The gap-tooth scheme for homogenization problems', *Multiscale Modeling and Simulation* **4**, 278–306.

Samaey, G., Roose, D. & Kevrekidis, I. G. (2006), 'Patch dynamics with buffers for homogenization problems', *J. Comput Phys.* **213**, 264–287.

Sieber, J., Marschler, C. & Starke, J. (2018), 'Convergence of Equation-Free Methods in the Case of Finite Time Scale Separation with Application to Deterministic and Stochastic Systems', *SIAM Journal on Applied Dynamical Systems* **17**(4), 2574–2614.