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

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

September 9, 2021

[*] 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 3.1.1).
- In 2D space adapt the code at the beginning of `configPatches2.m` for nonlinear diffusion (Section 3.7.1).
- In 3D space adapt the code at the beginning of `configPatches3.m` for wave propagation through a heterogeneous medium (Section 3.11.1), or the patches of the 3D heterogeneous diffusion of `homoDiffEdgy3.m` (Section 3.14).
- 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.

Chapter 4 provides small examples of how to parallelise the patch computations over multiple processors. But such parallelisation may be only useful for scenarios where the microscale code has many millions of operations per time-step.

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

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

# 2 Projective integration of deterministic ODEs

**Chapter contents**

## 2.1 Introduction

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

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

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

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

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

### Main functions

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

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

**Schematic for Projective Integration scheme**

**Set microsolver**
Code a function that interfaces to your 'black-box' microsolver, including the burst time, `bT`, of the microsolver. Possible aids:

- Use the Patch functions (Figure 3.1) to simulate a large-scale PDE, lattice, etc.
- Use `cmdc()` as a wrapper for the microsolver if the slow variables may change significantly over the microsolver burst.

**Set macro simulator**
Set the vector of output times `tSpan/Tspan`, and set initial values `x0`.

- If using `PIRKn()`, then intervals between times are the projective time-steps.
- If using `PIG()`, then intervals between times are as needed by `macroInt`.

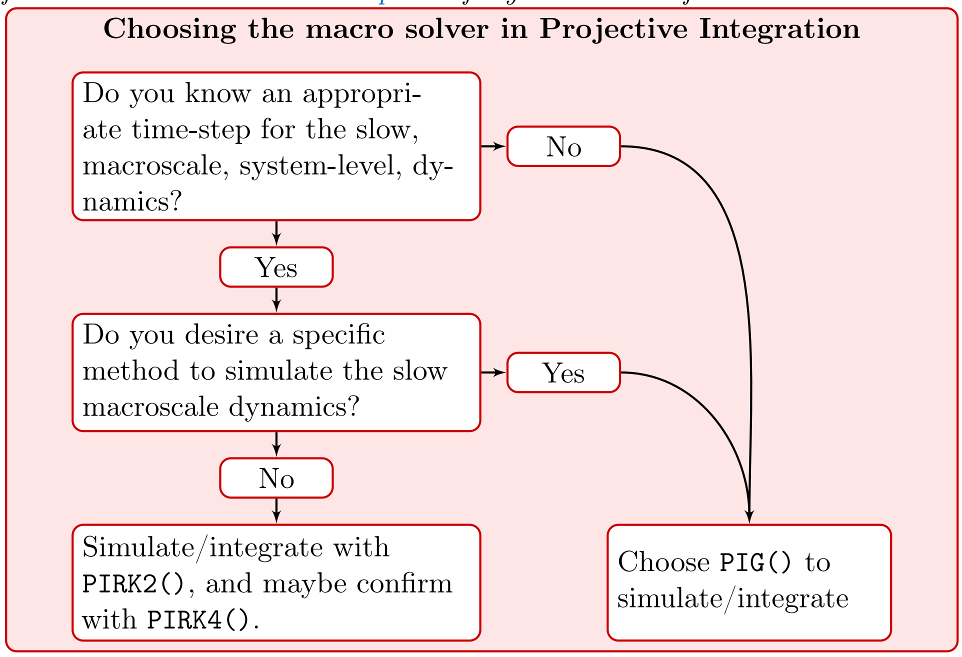
**Do Projective Integration** Invoke the appropriate Projective Integration function as, e.g., `x = PIRK2(microBurst, tSpan, x0)`, or `[t,x] = PIG(macroInt, microBurst, Tspan, x0)`. Additional optional outputs inform you of the microscale.

**Set lifting/ restriction** If needed, set functions `restrict()` and `lift()` to convert between macro and micro problems/ variables. These are optional arguments to the Projective Integration functions.

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

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

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



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

*Section contents*



### 2.2.1 Introduction

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

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

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

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

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

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

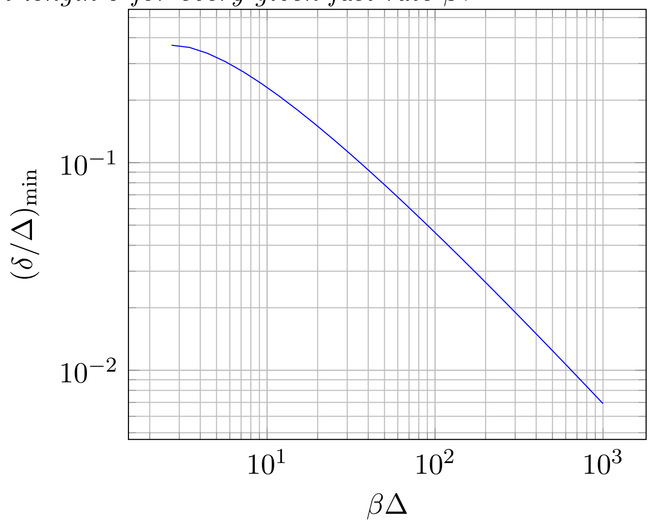
  Be wary that for very large scale separations (such as `MMepsilon<1e-5` in the Michaelis–Menten example), microscale integration by error-controlled variable-step routines (such as `ode23/45`) often generate microscale variations that ruin the projective extrapolation of `PIRK2()`. In such cases, a fixed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

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

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

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

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




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 $\varepsilon$ that you wish to achieve (e.g., $\varepsilon \approx 0.01$ for two digit accuracy); secondly, the slow dynamics of your system occurs at rate/ frequency of magnitude about $\alpha$; and thirdly, the rate of *decay* of your fast modes are faster than the lower bound $\beta$ (e.g., if three fast modes decay roughly like $e^{-12t}, e^{-34t}, e^{-56t}$ then $\beta \approx 12$). Then set

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

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

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

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

- `tms`, optional, is an $L$ dimensional column vector containing the microscale times within the burst simulations, each burst separated by `NaN`;
- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—each rows is an accurate estimate of the state at the corresponding

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

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

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

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

```
if nargin==0
```

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

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

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

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

```
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 $\epsilon$.

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

```
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 $2\epsilon$, and starting from the initial condition for the Michaelis–Menten system, at time $t = 0$, of $(x, y) = (1, 0)$ (off the slow manifold).

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

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

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

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

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

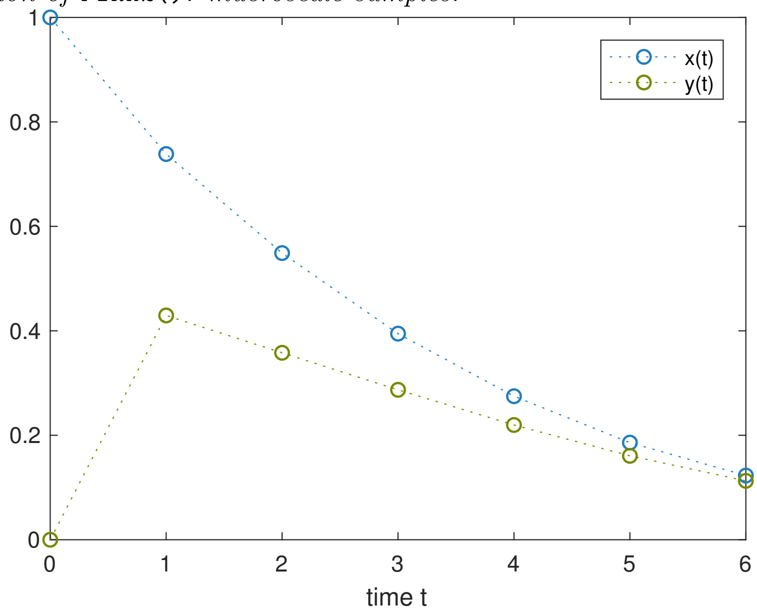



Figure 2.5 plots the macroscale and microscale results—also showing that the initial burst is by default twice as long. Observe the slow variable $x(t)$ is also affected by the initial transient (hence other schemes which 'freeze' slow variables are less accurate).

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

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

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

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

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

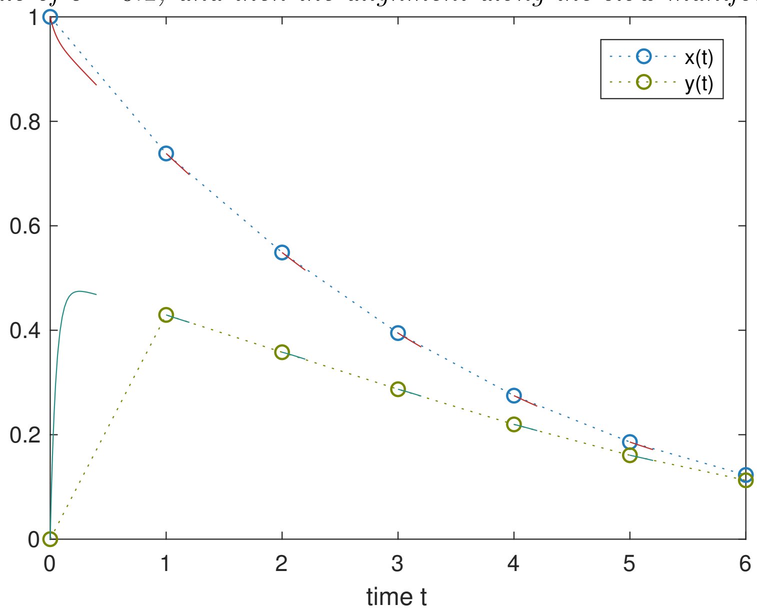
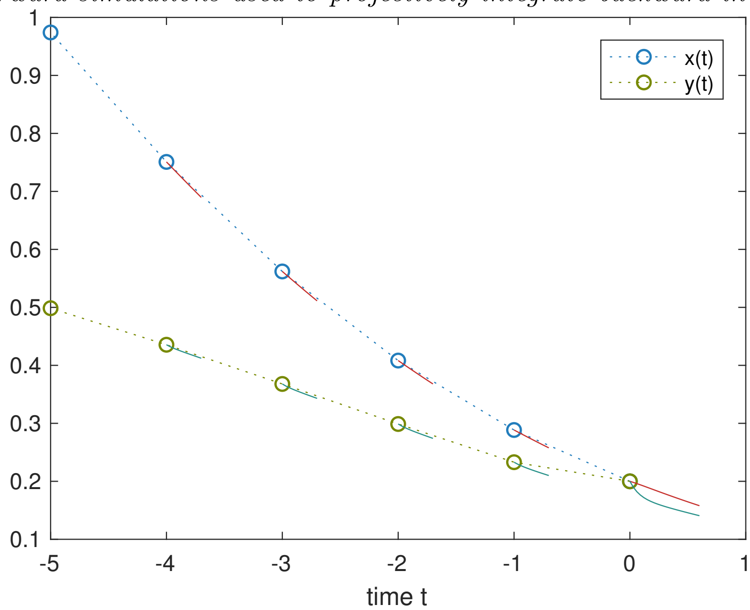



*Figure 2.6: Michaelis–Menten enzyme kinetics at* $\epsilon = 0.1$ *simulated backward with the projective integration of* `PIRK2()`*: the microscale bursts show the short forward simulations used to projectively integrate backward in time.*



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

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

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

*Section contents*



### 2.4.1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

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

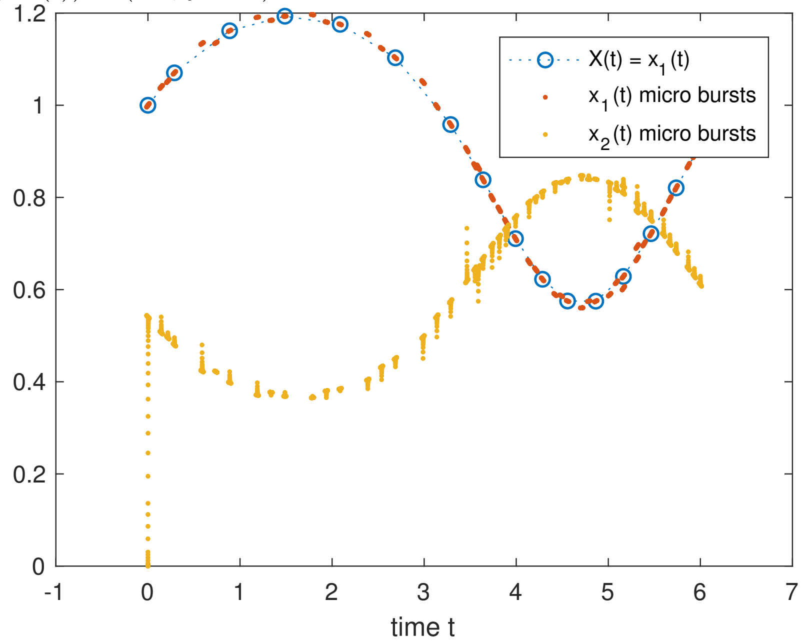
If the `cdmcFlag` is to be set without using a `restrict()` or `lift()` function, then use empty matrices `[]` for the restrict and lift functions.

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

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

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

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




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

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 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 $0 \leq t \leq 6$ from initial condition $\vec{x}(0) = (1, 0)$. You could set `Tspan=[0 -6]` to integrate backward in macroscale time with forward microscale bursts (Gear & Kevrekidis 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 $-5 \leq t \leq 0$ for parameter $\epsilon = 0.1$ . Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(|\Delta|/\epsilon)$ as here the macroscale time-step $\Delta = -1$.

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

```
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**

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

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

**Quick start** See Sections 3.1.1 and 3.7.1 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

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

**Patch scheme for spatio-temporal dynamics**

**Setup problem and construct patches**
Invoke `configpatches1` (for 1D) or `configpatches2` (for 2D) or `configpatches3` (for 3D) to setup the microscale problem (PDE, domain, boundary conditions, etc) and the desired patch structure (number of patches, patch size, coupling order, etc). These initialise the global struct `patches` that contains information required to simulate the microscale dynamics within each patch. If necessary, define additional components for the struct `patches` (e.g., see `homogenisationExample.m`).

**Simulate the multiscale system**
Generally invoke a PDE integrator to simulate the multiscale system of a user's microscale code within spatially separated patches. This integrator may be MATLAB/Octave defined (e.g., `ode23/ode45`) or user defined (e.g., a projective integrator).
Input to the integrator is the function `patchSys1` (for 1D) or `patchSys2` (for 2D) or `patchSys3` (for 3D) which interfaces to the microscale's code. Other inputs are the macro-time span and initial conditions. Output from the integrator is the solution field over the given time span, but only within the defined patches.

**Interface to microscale**
`patchSys1/2/3` interfaces with the microscale PDE/lattice system and invokes the patch coupling conditions. Input is the field in every patch at one time-step, and output is time-derivatives of the field, or values at the next time-step, as appropriate.

**Coupling conditions**
`patchEdgeInt1/2/3` (for 1D, 2D, or 3D respectively) couple patches together by setting edge-values via macroscale interpolation of order in global `patches.ordCC`.

**Microscale system**
A user's microscale code, pointed to by `patches.fun`, codes the microscale dynamics on the interior of the patch microgrids, `patches.x`, to compute either a micro-step or time-derivatives.

**Process results and plot**

## 3.1 `configPatches1()`: configures spatial patches in 1D

*Section contents*



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

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

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

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, that computes time derivatives (or time-steps) of quantities on the 1D micro-grid within all the 1D patches.
- `Xlim` give the macro-space spatial domain of the computation: 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 spatial patches.
- `ordCC`, must be $\geq -1$, is the 'order' of interpolation across empty space of the macroscale patch values to the edge of the patches for inter-patch coupling: where `ordCC` of 0 or $-1$ gives spectral interpolation; and `ordCC` being odd specifies staggered spatial grids.
- `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 centre-patch lattice point.
- `nEdge` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).
- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right edge-values from right/left next-to-edge values. If false or omitted, then interpolate from centre-patch values.
- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- `hetCoeffs`, *optional*, default empty. Supply a 1/2D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times n_c$, where $n_c$ is the number of different sets of coefficients. The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the $n_c$ coefficients being $n_c$ parameters in some macroscale formula.
  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the first point in each patch.
  - If `nEnsem` $> 1$ (value immaterial), then reset `nEnsem` $:= m_x$ and construct an ensemble of all $m_x$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities, then this coupling cunningly preserves symmetry .
- `nCore`, *optional-experimental*, default one, but if more, and only for non-EdgyInt, then interpolates from an average over the core of a patch, a core of size ??. Then edge values are set according to interpolation of the averages?? or so that average at edges is the interpolant??
- `'parallel'`, true/false, *optional*, default=false. If false, then all patch computations are on the user's main CPU—although a user may well separately invoke, say, a GPU to accelerate sub-patch computations.

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

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

```
if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)`, 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.

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

- `.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 $x_{iI}$ of the $i$th microscale grid point in the $I$th patch.
- `.ratio` is the size ratio of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times n_c$ 2D array of microscale heterogeneous coefficients, or
  - if `nEnsem` $> 1$, (`nSubP(1)` $- 1) \times n_c \times m_x$ 3D array of $m_x$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

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

```
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 $\leftrightarrow$ patchSys1 $\leftrightarrow$ user's PDE
3. process results

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

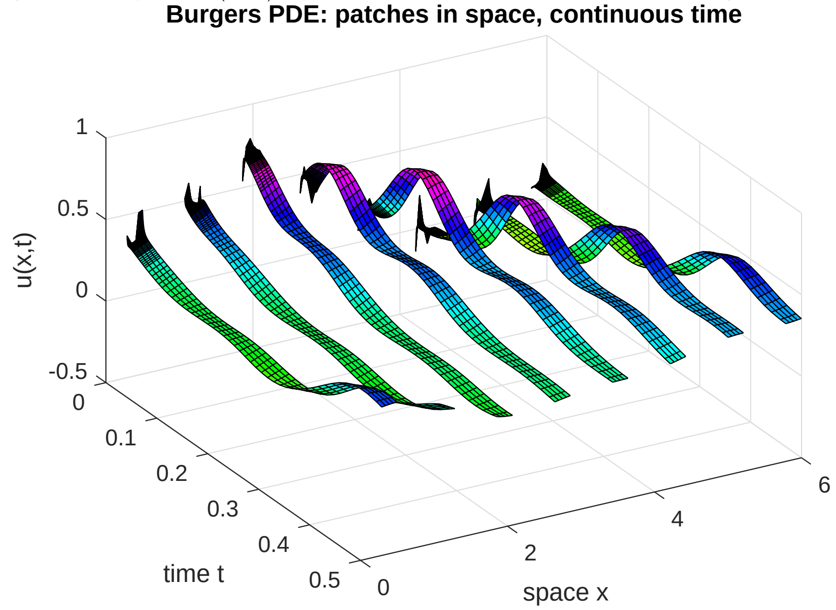
```
global patches
patches = configPatches1(@BurgersPDE,[0 2*pi], nan, 8, 0, 0.2, 7);
```

Set some 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 3.2).

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




```
if ~exist('OCTAVE_VERSION','builtin')
[ts,us] = ode15s( @patchSys1,[0 0.5],u0(:));
else % octave version
[ts,us] = odeOcts(@patchSys1,[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(0.8*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 patchSys1()).

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

```
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
```

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

To simulate in time with 1D spatial patches we often need to interface a user's time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. It mostly assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.1) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

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

#### Input

- `u` is a vector/array of length $\texttt{nSubP}\cdot\texttt{nVars}\cdot\texttt{nEnsem}\cdot\texttt{nPatch}$ where there are $\texttt{nVars}\cdot\texttt{nEnsem}$ field values at each of the points in the $\texttt{nSubP}\times\texttt{nPatch}$ grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches1()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches)` that computes the time derivatives on the patchy lattice. The array `u` has size $\texttt{nSubP}\times\texttt{nVars}\times\texttt{nEnsem}\times\texttt{nPatch}$. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - `.x` is $\texttt{nSubP}\times 1\times 1\times\texttt{nPatch}$ array of the spatial locations $x_i$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

#### Output

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

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

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value (Roberts 2003, Roberts & Kevrekidis 2007), or the patch-core average (Bunder et al. 2017), or the opposite next-to-edge values (Bunder et al. 2020)—this last alternative often maintains symmetry. This function is primarily used by `patchSys1()` but is also useful for user graphics. When using core averages, assumes the averages are sensible macroscale variables: then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). [2]

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

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

### Input

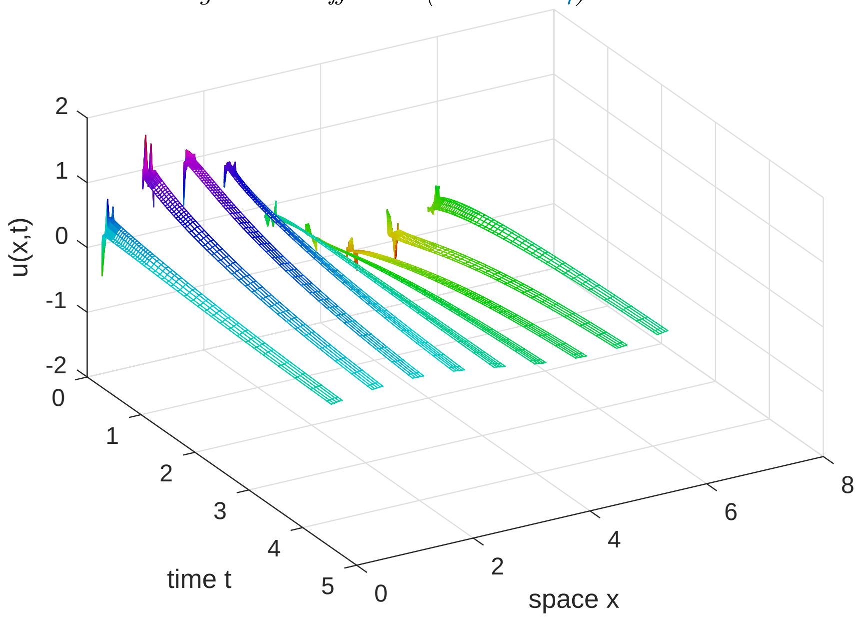
- `u` is a vector/array of length `nSubP`·`nVars`·`nEnsem`·`nPatch` where there are `nVars`·`nEnsem` field values at each of the points in the `nSubP`×`nPatch` multiscale spatial grid.
- `patches` a struct largely set by `configPatches1()`, and which includes the following.
  - `.x` is `nSubP` $\times 1 \times 1 \times$ `nPatch` array of the spatial locations $x_{iI}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation, integer $\geq -1$.
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation, and zero for ordinary grid.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation.
  - `.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.
  - `.parallel` whether serial or parallel.

### Output

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

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

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




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

*Section contents*



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

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

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

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

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

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

### 3.4.1 Script to simulate via stiff or projective integration

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

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

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

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

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

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

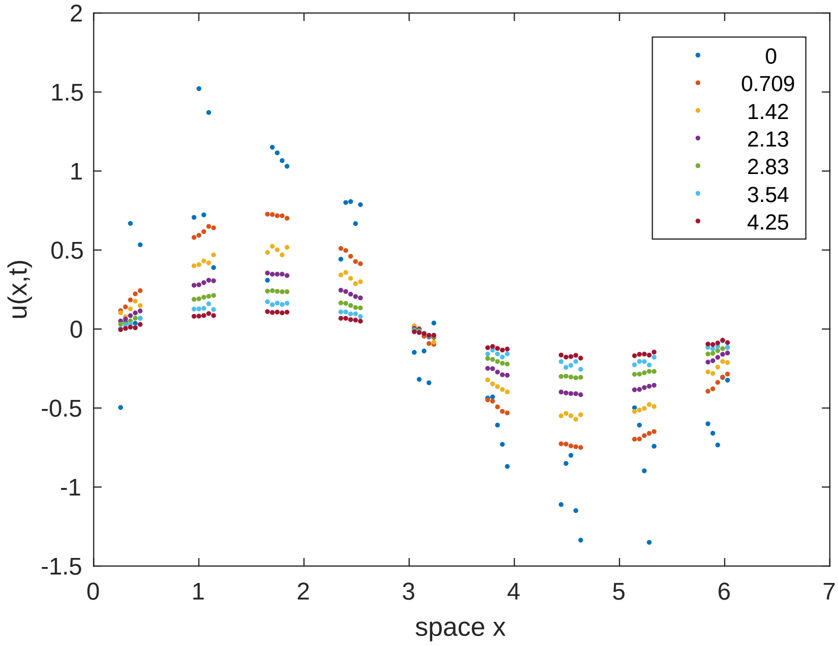
Plot the simulation in Figure 3.3.

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

The code may invoke this integration interface.

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

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




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

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

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

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

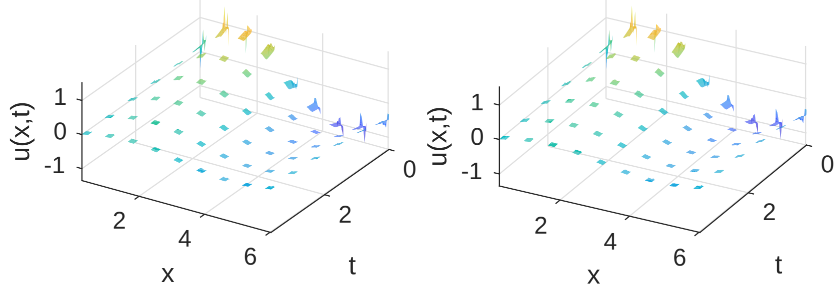
Mark that edge of patches are not to be used in the projective extrapolation by setting initial values to NaN.

```
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
```

*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.*




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

Plot the macroscale predictions to draw Figure 3.4.

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

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

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

End of this example script.

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

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

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

```
        buu = patches.cs(:,2,:).*u.^2;
        ut(i,:) = ut(i,:)-(buu(i+1,:)-buu(i-1,:))/(dx*2);
  end
end% function
```

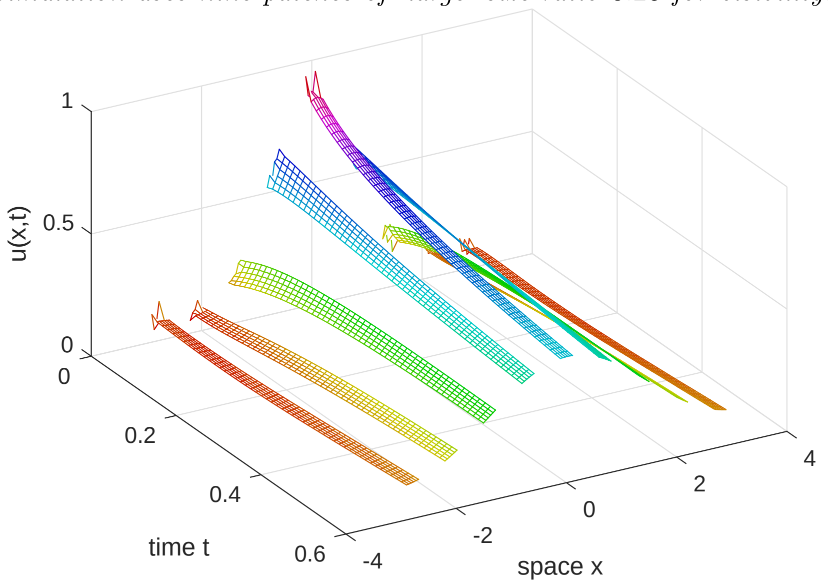
### 3.4.3 `heteroBurst()`: a burst of heterogeneous diffusion

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

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

Fin.

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




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

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

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

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

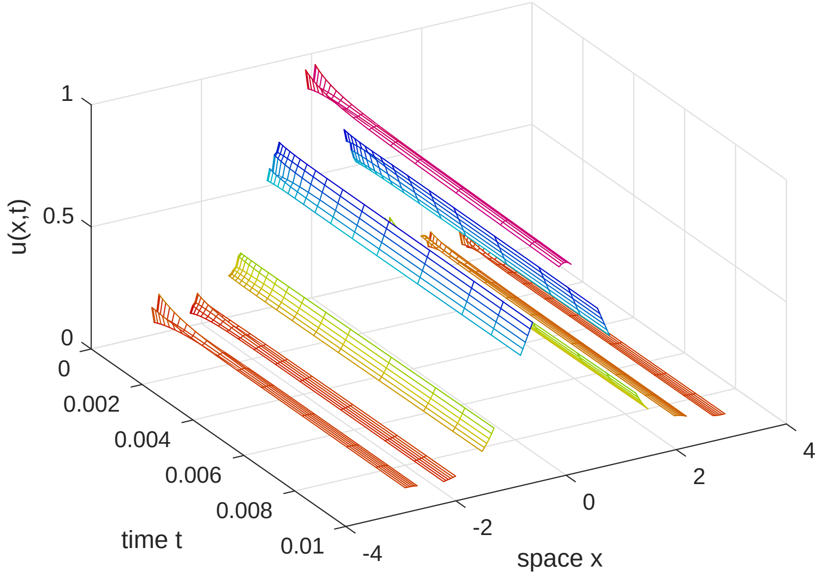
in terms of the centred difference operator $\delta$. The system has a microscale heterogeneity via the coefficients $c_{i+1/2}$ which we assume to have some given known periodicity. Figure 3.6 shows one patch simulation of this system: observe the effects of the heterogeneity within each patch.

### 3.5.1 Script code to simulate heterogeneous diffusion systems

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

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

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




4. use patchSys1 to explore the Jacobian

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

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

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

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

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

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

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

Integrate using standard integrators.

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

**Plot space-time surface of the simulation** We want to see the edge values of the patches, so we adjoin a row of `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=patchSys1(0,u0);
    Jac(:,j)=dudt(i);
  end
  nonSymmetric=norm(Jac-Jac')
  assert(nonSymmetric<5e-9,'failed symmetry')
  Jac(abs(Jac)<1e-12)=0;
```

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

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

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

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

```
  [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.

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

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

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

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

Fin.

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

The Landau–Lifshitz equation describes the precessional motion of magnetization $\vec{M}$ in a solid (see *Landau–Lifshitz–Gilbert equation* in Wikipedia). In a medium with microscale heterogeneity $a(x)$, and with phenomenological damping parameter $\alpha$, we explore the dynamics of $\vec{M}(x,t)$ governed by the nonlinear Landau–Lifshitz PDE (Leitenmaier & Runborg 2021, (1.1)) [3]

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

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

Figure 3.8 shows an example simulation in time generated by the patch scheme applied to the above Landau–Lifshitz PDE on the spatial domain $[0,1]$ with domain boundary conditions of 1-periodicity. The inter-patch coupling is realised by interpolation of the patch's next-to-edge values to the patch opposite edges. Such coupling preserves symmetry in many systems (quartic interpolation appears to be the lowest order that generally gives good accuracy). With damping parameter $\alpha = 0.001$ then the largest few macroscale modes decay with rate roughly 0.1, and so are negligibly damped over a time of 0.1.

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

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

in terms of the centred difference operator $\delta$. The system has a microscale heterogeneity via the coefficients $a_{i+1/2}$ which we assume to have some given known periodicity (Leitenmaier & Runborg 2021, pp.6,27). Figure 3.8 shows a patch simulation of this system: observe the effects of the heterogeneity within each patch.

**Parameters** There are two closely related examples (Leitenmaier & Runborg 2021, pp.6,27), that we distinguish here with parameter `ex5p1`: set to either zero or one. The Landau–Lifshitz dissipation parameter $\alpha$ should be small. If the initial conditions are smooth, then `ode15s` has no problems for $\alpha = 0.001$. [4]

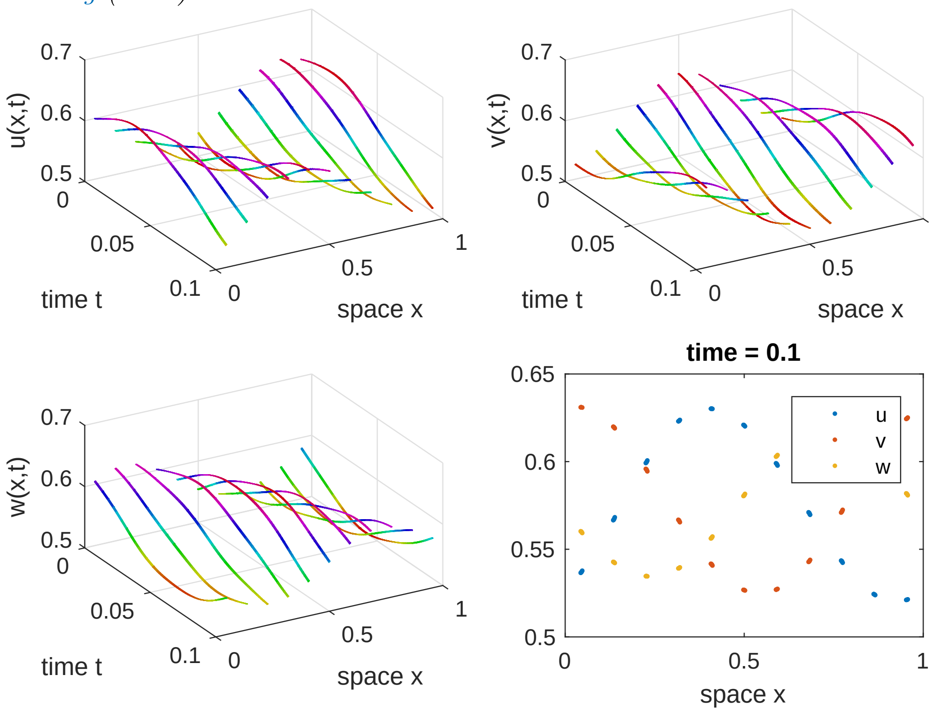
```
global alpha ex5p1
ex5p1 = 0; % set to 1 for L&O example of p.27
alpha = 0.001 % phenomenological damping parameter
```

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

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

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

*Figure 3.8: magnetic field $\vec{M}(x,t) = (u,v,w)$ of the gap-tooth scheme applied to the heterogeneous Landau–Lifshitz* PDE *to show the emergent macroscale wave-like dynamics. This simulation uses eleven patches in space of size ratio* 0.055*. Compare the time $t = 0.1$ graph with Fig. 2.1 of Leitenmaier & Runborg (2021).*




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

### 3.6.1 Script code to simulate heterogeneous diffusion systems

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

1. configPatches1
2. ode15s ↔ patchSys1 ↔ heteroLanLif1D
3. plot the simulation

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

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

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

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

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

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

Integrate using standard integrators.

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

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

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

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

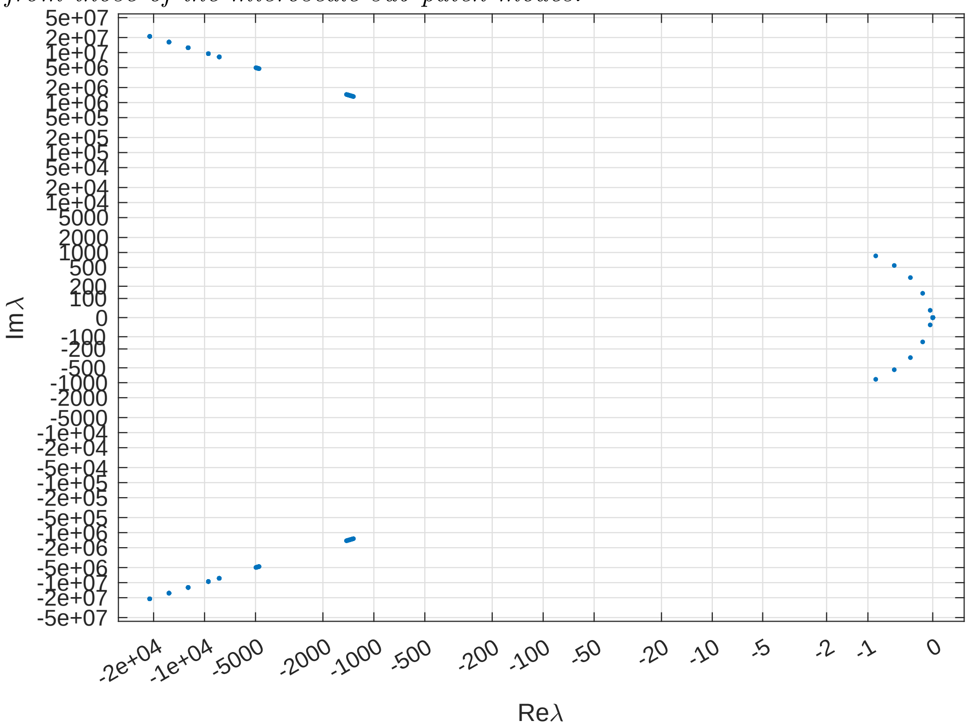
```
Mabs = sqrt( sum(Ms.^2,3) );
meanMabs = mean(Mabs(:),'omitnan')
stdevMabs = std(Mabs(:),'omitnan')
```

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

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

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

*Figure 3.9: spectrum of eigenvalues of the multiscale patch scheme applied to the Landau–Lifshitz* PDE*. The macroscale eigenvalues are clearly separated from those of the microscale sub-patch modes.*




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

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

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

### 3.6.2 Spectrum of the coded patch system

It appears the spectrum has the following properties as shown by Figure 3.9, with $N = \texttt{nPatch}$ and $n = \texttt{nSubP} - 2$, and on base of $\vec{M} = \vec{1}/\sqrt{3}$.

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

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

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

Form the Jacobian by numerical differentiation.

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

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

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

Plot the spectrum of eigenvalues on quasi-log axes.

```
figure(2),clf
scr=1; sci=1e-2;
plot(asinh(scr*real(eval)),asinh(sci*imag(eval)),'.')
xlabel('Re\lambda'), ylabel('Im\lambda')
ticks=[1;2;5]*10.^(0:6);
ticks=sort([0;ticks(:);-ticks(:)]);
set(gca,'Xtick',asinh(ticks) ...
    ,'XtickLabel',cellstr(num2str(ticks/scr,4)) ...
```

```
    ,'XTickLabelRotation',30)
set(gca,'Ytick',asinh(ticks) ...
    ,'YtickLabel',cellstr(num2str(ticks/sci,4)))
grid
axis tight; lims=axis; dl=diff(lims);
axis(lims+[-dl(1) +dl(1) -dl(3) +dl(3)]*0.04)
ifOurCf2eps([mfilename 'Spec'])
```

### 3.6.3 `heteroLanLif1D()`: heterogeneous Landau–Lifshitz PDE

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

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

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

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

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

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

```
  i = 2:size(M,1)-1;   % interior points in a patch
  MH=nan+H; % preallocate for MxH
  MH(:,3,:,:) = M(i,1,:,:).*H(:,2,:,:)-M(i,2,:,:).*H(:,1,:,:);
  MH(:,2,:,:) = M(i,3,:,:).*H(:,1,:,:)-M(i,1,:,:).*H(:,3,:,:);
  MH(:,1,:,:) = M(i,2,:,:).*H(:,3,:,:)-M(i,3,:,:).*H(:,2,:,:);
  MMH=nan+H; % preallocate for MxMxH
  MMH(:,3,:,:)= M(i,1,:,:).*MH(:,2,:,:)-M(i,2,:,:).*MH(:,1,:,:);
  MMH(:,2,:,:)= M(i,3,:,:).*MH(:,1,:,:)-M(i,1,:,:).*MH(:,3,:,:);
  MMH(:,1,:,:)= M(i,2,:,:).*MH(:,3,:,:)-M(i,3,:,:).*MH(:,2,:,:);
  Mt = nan+M; % preallocate output array
  Mt(i,:,:,:) = -MH-alpha*MMH;
end% function
```

Fin.

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

*Section contents*



Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys2()`. Section 3.7.1 lists an example of its use.

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

**Input** If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 3.7.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, 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)`] × [`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 specified rectangular domain.
- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` gives the number of patches ($\geq 1$) in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the edge-values of the patches: currently must be $0, 2, 4, \ldots$; 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 of the two directions.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise `nSubP(1:2)` gives the number in each direction. If not using `EdgyInt`, then must be odd so that there is/are centre-patch micro-grid point/ lines in each patch.

- `nEdge` (not yet implemented), *optional*, default=1, for each patch, the number of edge values set by interpolation at the edge regions of each patch. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

- `EdgyInt`, true/false, *optional*, default=false. If true, then interpolate to left/right/top/bottom edge-values from right/left/bottom/top next-to-edge values. If false or omitted, then interpolate from centre-patch lines.

- `nEnsem`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- `hetCoeffs`, *optional*, default empty. Supply a 2/3D array of microscale heterogeneous coefficients to be used by the given microscale `fun` in each patch. Say the given array `cs` is of size $m_x \times m_y \times n_c$, where $n_c$ is the number of different sets of coefficients. For example, in heterogeneous diffusion, $n_c = 2$ for the diffusivities in the *two* different spatial directions (or $n_c = 3$ for the diffusivity tensor). The coefficients are to be the same for each and every patch; however, macroscale variations are catered for by the $n_c$ coefficients being $n_c$ parameters in some macroscale formula.

  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1)$-point in each patch.

  - If `nEnsem` $> 1$ (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y$ and construct an ensemble of all $m_x \cdot m_y$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in $x$ and $y$ directions, respectively, then this coupling cunningly preserves symmetry.

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

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

**Output** The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[5]

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

```
if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)`, 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 $x_{iI}$ of the microscale grid points in every patch.
- `.y` (6D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `nPatch(2)` array of the regular spatial locations $y_{jJ}$ of the microscale grid points in every patch.
- `.ratio` $1 \times 2$, are the size ratios of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri`, `.bo`, `.to` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)` $- 1) \times$ (`nSubP(2)` $- 1) \times n_c$ 3D array of microscale heterogeneous coefficients, or
  - if `nEnsem` $> 1$, (`nSubP(1)` $- 1) \times$ (`nSubP(2)` $- 1) \times n_c \times m_x m_y$ 4D array of $m_x m_y$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

### 3.7.1 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 $\leftrightarrow$ patchSys2 $\leftrightarrow$ user's PDE
3. process results

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

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

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

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

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

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

Start by showing the initial conditions of Figure 3.10 while the simulation computes.

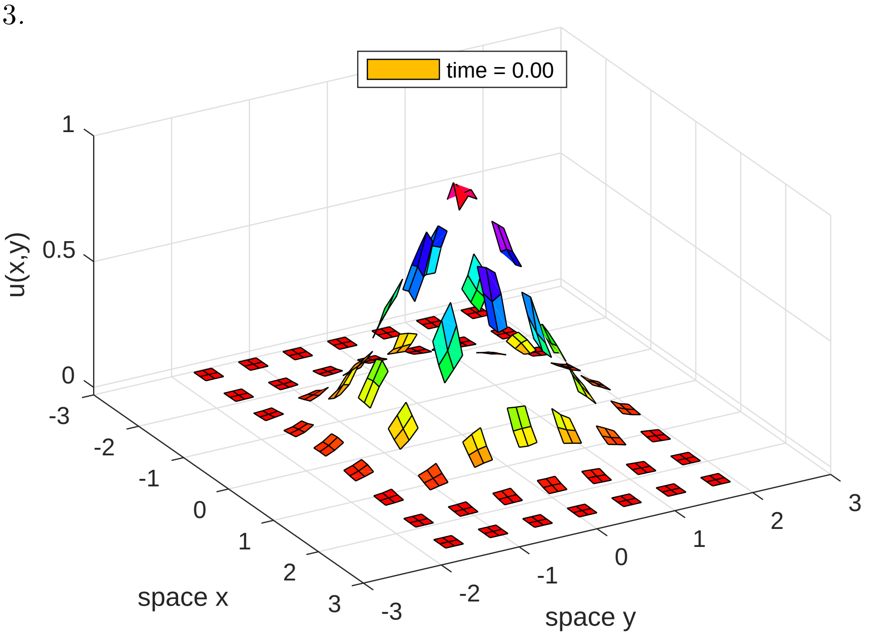
```
u = reshape(permute(squeeze(u0) ...
    ,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = surf(x(:),y(:),u');
axis([-3 3 -3 3 -0.03 1]), view(60,40)
legend('time = 0.00','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
colormap(hsv)
ifOurCf2eps([mfilename 'ic'])
```

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

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

Animate the computed simulation to end with Figure 3.11. Use `patchEdgeInt2` to interpolate patch-edge values (but not corner values, and even if not drawn).

*Figure 3.10: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 3.11 plots the computed field at time $t = 3$.*




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

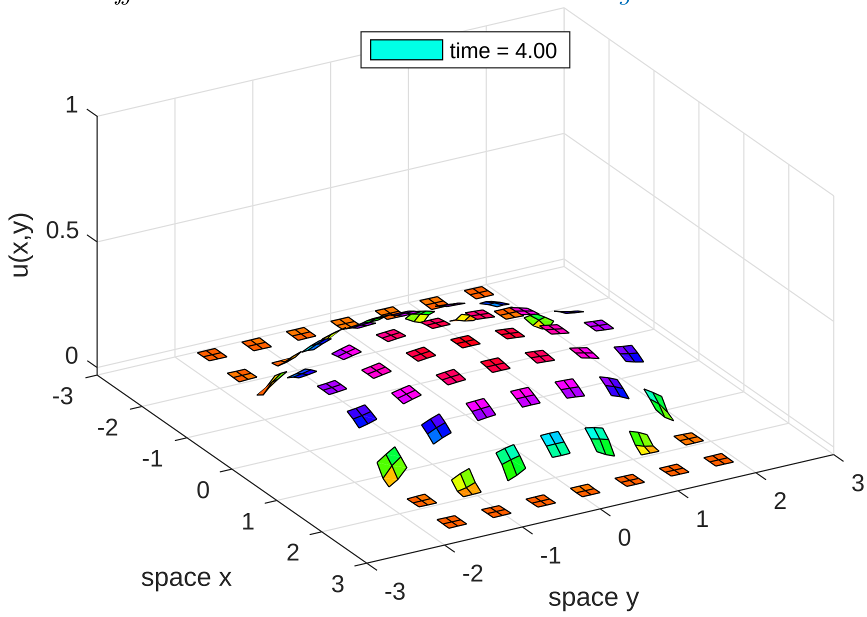
Upon finishing execution of the example, exit this function.

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

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

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

*Figure 3.11: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 3.10.*




```
end
```

## 3.8 patchSys2(): interface 2D space to time integrators

To simulate in time with 2D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.7) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

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

### Input

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

### Output

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

## 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. Research (Roberts et al. 2014, Bunder et al. 2019) indicates the patch centre-values are sensible macroscale variables, and macroscale interpolation of these determine patch-edge values. However, for computational homogenisation in multi-D, interpolating patch next-to-edge values appears better (Bunder et al. 2020). This function is primarily used by `patchSys2()` but is also useful for user graphics. [6]

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

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

### Input

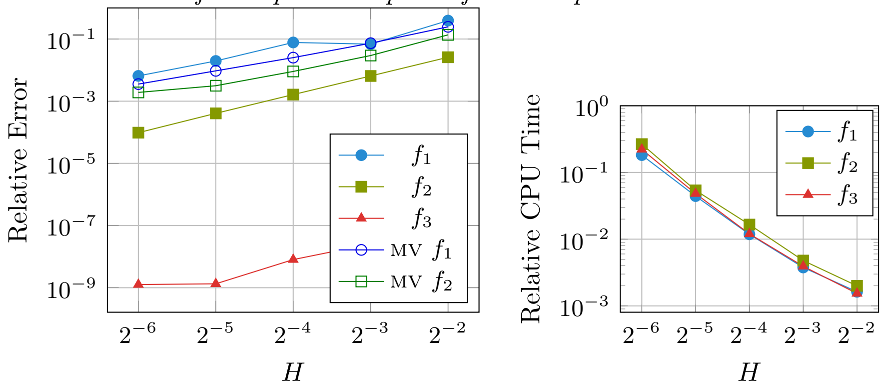
- `u` is a vector/array of length `prod(nSubP)`·`nVars`·`nEnsem`·`prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP1` · `nSubP2` · `nPatch1` · `nPatch2` multiscale spatial grid on the `nPatch1` · `nPatch2` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is `nSubP1`$\times 1 \times 1 \times 1 \times$`nPatch1`$\times 1$ array of the spatial locations $x_{iI}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times$ `nPatch2` array of the spatial locations $y_{jJ}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in both the $x, y$-directions.
  - `.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.
  - `.parallel` whether serial or parallel.

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

### Output

- u is 6D array, nSubP1 · nSubP2 · nVars · nEnsem · nPatch1 · nPatch2, of the fields with edge values set by interpolation (and corner vales set to NaN).

*Figure 3.12: results for the computational homogenisation of a forced, non-autonomous, 2D wave (Section 3.10). (left) relative* RMS *error of the patch scheme, each patch of width* $1/128$*, as a function of patch spacing* $H$*. The unfilled symbols are those of the energy norm from Maier & Verfürth (2021) (their Figure 5.1). (right) the relative compute time decreases very quickly in* $H$ *as there are fewer patches spaced further apart.*




## 3.10 `homoWaveEdgy2`: computational homogenisation of a forced, non-autonomous, 2D wave via simulation on small patches

This section extends to 2D waves, in a microscale heterogeneous media, the 2D diffusion code discussed in **??**. It favourably compares to the examples of Maier & Verfürth (2021).

Figure 3.12 summarises the results here. The left (larger) graph shows the error in the patch scheme decreasing with decreasing patch spacing $H$ (increasing number of patches). Forcing $f_1$ and $f_2$ are as specified by §5.1 of Maier & Verfürth (2021), whereas $f_3$ here is $f$ in their §5.2. For the case of forcing $f_1$ which is discontinuous in space (at $x = 0.4$), the errors are similar to that of Maier & Verfürth (2021)—compare the filled with unfilled circles. For the case of forcing $f_2$ which is continuous in the spatial domain, except for a second derivative discontinuity in its odd-periodic extension, the errors of the patch scheme are an order of magnitude better that that of Maier & Verfürth (2021)—compare the filled with unfilled squares. For the case of forcing $f_3$ which is smooth in the domain and in its odd-periodic extension, the patch scheme errors, roughly $10^{-8}$, are at the tolerance of the time integration. Two caveats in a comparison with Maier & Verfürth (2021) are the slightly different norms used, and that they also address errors in the time integration, whereas here we use a standard adaptive integrator in order to focus purely on the spatial errors of the patch scheme.

Now let's code the simulation of the forced, non-autonomous, 2D wave. Maier & Verfürth (2021) have Dirichlet BCs of zero around the unit square, so replicate here by the odd periodic extension to the spatial domain $[-1, 1]^2$. In their §5.1, their microscale mesh step is $1/512 = 2^{-9}$. Coding that here results in a compute time of roughly 90 minutes, so here I provide a much coarser case that computes in only a few minutes: change as you please.

```
clear all
dx = 1/128  % 1/512=2^{-9} is the original, but takes 90 mins
```

The heterogeneity is of period four on the microscale lattice, so code a minimal patch size that covers one period.

```
epsilon = 4*dx
nPeriodsPatch = 1
mPeriod = round(epsilon/dx)
nSubP = mPeriod*nPeriodsPatch+2
```

Choose which of three forcing functions to use

```
fn=2
```

Maier & Verfürth (2021) use varying number of macroscale grid steps from 4 to 64 on $[0, 1]$ so here on $[-1, 1]$ we use double the number patches in each direction. Loop over the number of patches used, starting with the full domain simulation, and then progressively coarsening the macroscale grid of patches.

```
nPatch = 2/epsilon/nPeriodsPatch
for iPat=0:9
if iPat>0, nPatch=nPatch/2, end
if nPatch<8, break, end
```

Set the periodic heterogeneous coefficient, isotropic:

$$a_\epsilon(t, x) = \big[3 + \sin(2\pi x/\epsilon) + \sin(2\pi t)\big] \cdot \big[3 + \sin(2\pi y/\epsilon) + \sin(2\pi t)\big],$$

which being in product form with two time-dependencies we store as the two spatially varying factors—although to preserve odd symmetry we phase shift the heterogeneity from sines to cosines. It is a user's choice whether to code such spatial dependencies here with `cHetr` or within the time derivative function itself. In this case, I choose to code microscale heterogeneous coefficients here via `cHetr`, and the macroscale variation of $f_i$ in the time derivative function.

Here the period of the heterogeneity is only four microscale lattice points in each direction (which is pretty inaccurate on the microscale, but immaterial as we and Maier & Verfürth (2021) only compare to the coded system on the microscale lattice, not to the PDE). With the following careful choices we ensure all the hierarchy of patch schemes both maintain odd symmetry, and also compute on grid points that are common with the full domain.

```
ratio = (nSubP-2)*dx/(2/nPatch)
Xleft=(1-ratio)/nPatch;
xmid=Xleft+dx*(0:mPeriod-1)';    % half-points
xi = Xleft+dx*(-0.5:mPeriod-1)'; % grid-points
% two components for ax, the x-dirn interactions
cHetr(:,:,1) = (3+cos(2*pi*xmid/epsilon))+0*xi';
cHetr(:,:,2) = 0*xmid+(3+cos(2*pi*xi'/epsilon));
% two components for ay, the y-dirn interactions
cHetr(:,:,3) = (3+cos(2*pi*xi/epsilon))+0*xmid';
cHetr(:,:,4) = 0*xi+(3+cos(2*pi*xmid'/epsilon));
```

Configure patches using spectral interpolation. Quadratic interpolation did not seem significantly different for the case of discontinuous forcing $f_1$.

```
configPatches2(@heteroWave2,[-1 1 -1 1],nan,nPatch ...
    ,0,ratio,nSubP ,'EdgyInt',true ,'hetCoeffs',cHetr );
```

A check on the spatial geometry.

```
global patches
dxPat=diff(patches.x(1:2));
assert(abs(dx-dxPat)<1e-9,"dx mismatch")
```

**Simulate** Set the particular forcing function to use, and the zero initial conditions of a simulation.

```
patches.eff=fn;
clear uv0
uv0(:,:,1,1,:,:) = 0*patches.x+0*patches.y;
uv0(:,:,2,1,:,:) = 0*patches.x+0*patches.y;
```

Integrate using standard integrators. Maier & Verfürth (2021) use a scheme with fixed time-step of $\tau = 2^{-7} = 1/128$ . Here `ode23` uses variable steps of about $0.0003$ , and takes 7 s for `nPatch=2*4` (whereas `ode15s` takes 149 s—even for the dissipating case), and takes 287 s for `nPatch=2*32` and roughly 4000 s for full domain `nPatch=2*128`.

```
disp('Now simulate over time')
tic
[ts,us] = ode23(@patchSys2, linspace(0,1,11), uv0(:));
if iPat==0,  odeTime0=toc
else relOdeTime(iPat)=toc/odeTime0
end
```

**Compute error compared to full domain simulation** Get spatial coordinates of patch-interior points, and reshape to column vectors.

```
i = 2:nSubP-1;
x = squeeze(patches.x(i,:,:,:,:,:));
y = squeeze(patches.y(:,i,:,:,:,:));
x=x(:); y=y(:);
```

At the final time of $t = 1$ , get the row vector of data, form into the 6D array via the interpolation to the edges, and reshape patch-interior points to 2D spatial array.

```
uv = squeeze( patchEdgeInt2(us(end,:)));
u = squeeze( uv(i,i,1,:,:) );
u = reshape(permute(u,[1 3 2 4]),[numel(x) numel(y)]);
```

If this is the full domain simulation, then store as the reference solution.

```
if iPat==0
  x0=x; y0=y; u0=u;
```

```
  rms0=sqrt(mean(u0(:).^2))
else
```

Else compute the error compared to the full domain solution. First find the indices of the full domain that match the spatial locations of the patch scheme.

```
  [i,k] = find(abs(x0-x')<1e-9);
  assert(length(i)==length(x),'find error in index i')
  [j,k] = find(abs(y0-y')<1e-9);
  assert(length(j)==length(y),'find error in index j')
```

The RMS error over the surface is

```
  errs=u-u0(i,j);
  relrmserr(iPat)=sqrt(mean(errs(:).^2))/rms0
  H(iPat)=2/nPatch
end%if iPat
```

End the loop over the various number of patches, and return. Further, here not executed, code in the file animates the solution over time, and computes spectrum of the system.

```
end%for iPat
figure(1), clf
loglog(H,relrmserr,'o:'),  grid on
xlabel('H'), ylabel('relative error')
return
```

### 3.10.1 `heteroWave2()`: heterogeneous Waves

This function codes the lattice heterogeneous waves inside the patches. The forced wave PDE is

$$u_t = v, \quad v_t = \vec{\nabla}(a\vec{\nabla} \cdot u) + f$$

for scalars $a(t,x,y)$ and $f(t,x,y)$ where $a$ has microscale variations. For 6D input arrays `u`, `x`, and `y` (via edge-value interpolation of `patchSys2`, Section 3.8), computes the time derivative at each point in the interior of a patch, output in `ut`. The four 2D arrays of heterogeneous interaction coefficients, $c_{ijk}$, have previously been stored in `patches.cs` (3D).

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

```
function ut = heteroWave2(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps, and interior point indices.

```
  dx = diff(patches.x(2:3));  % x micro-scale step
  dy = diff(patches.y(2:3));  % y micro-scale step
  i = 2:size(u,1)-1; % x interior points in a patch
  j = 2:size(u,2)-1; % y interior points in a patch
  assert(max(abs(u(:)))<9999,"u-field exploding")
```

Form coefficients here—odd periodic extension. To avoid slight errors in periodicity (in full domain simulation), first adjust any coordinates crossing $x = \pm 1$ or $y = \pm 1$.

```
x=patches.x; y=patches.y;
l=find(abs(x)>1); x(l)=x(l)-sign(x(l))*2;
l=find(abs(y)>1); y(l)=y(l)-sign(y(l))*2;
```

Then set at this time three possible forcing functions, although only use one depending upon `patches.eff`. Forcing $f_1$ and $f_2$ are as specified by §5.1 of Maier & Verfürth (2021), whereas $f_3$ here is $f$ in their §5.2.

```
f1 = ( (abs(x)>0.4)*(20*t+230*t^2) ...
      +(abs(x)<0.4)*(100*t+2300*t^2) ).*sign(x).*sign(y);
f2 = 20*t*x.*(1-abs(x)).*y.*(1-abs(y)) ...
    +230*t^2*(sign(y).*x.*(1-abs(x))+sign(x).*y.*(1-abs(y)));
f3 = (5*t+50*t^2)*sin(pi*x).*sin(pi*y);
```

Also set the heterogeneous interactions at this time.

```
ax = (patches.cs(:,:,1)+sin(2*pi*t)) ...
   .*(patches.cs(:,:,2)+sin(2*pi*t));
ay = (patches.cs(:,:,3)+sin(2*pi*t)) ...
   .*(patches.cs(:,:,4)+sin(2*pi*t));
```

Reserve storage (using `nan+u` appears quickest), and then assign time derivatives for interior patch values due to the heterogeneous interaction and forcing.

```
  ut = nan+u;  % preallocate output array
  ut(i,j,1,:) = u(i,j,2,:);
  ut(i,j,2,:) ...
  = diff(ax(:,j).*diff(u(:,j,1,:),1),1)/dx^2 ...
   +diff(ay(i,:).*diff(u(i,:,1,:),1,2),1,2)/dy^2 ...
   +(patches.eff==1)*f1(i,j,:,:) ...
   +(patches.eff==2)*f2(i,j,:,:) ...
   +(patches.eff==3)*f3(i,j,:,:) ...
   + 1e-4*(diff(u(:,j,2,:),2,1)/dx^2+diff(u(i,:,2,:),2,2)/dy^2);
end% function
```

In the last line above, the slight damping of $10^{-4}$ causes microscale modes to decay at rate $e^{-28t}$, with frequencies 2000–5000, whereas macroscale modes decay with rates roughly 0.0005–0.05 with frequencies 10–100. This slight damping term may correspond to the weak damping of the backward Euler scheme adopted by Maier & Verfürth (2021) for time integration.

## 3.11 `configPatches3()`: configures spatial patches in 3D

*Section contents*



Makes the struct `patches` for use by the patch/gap-tooth time derivative/step function `patchSys3()`, and possibly other patch functions. Sections 3.11.1 and 3.14 list examples of its use.

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

**Input** If invoked with no input arguments, then executes an example of simulating a heterogeneous wave PDE—see Section 3.11.1 for the example code.

- `fun` is the name of the user function, `fun(t,u,patches)` or `fun(t,u)`, 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 rectangular cuboid [`Xlim(1)`,`Xlim(2)`] × [`Xlim(3)`,`Xlim(4)` × [`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 specified rectangular domain.
- `nPatch` sets the number of equi-spaced spatial patches: if scalar, then use the same number of patches in all three directions, otherwise `nPatch(1:3)` gives the number ($\geq 1$) of patches in each direction.
- `ordCC` is the 'order' of interpolation for inter-patch coupling across empty space of the macroscale patch values to the 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 all three directions, otherwise `ratio(1:3)` gives the ratio in each of the three directions.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in all three directions, otherwise `nSubP(1:3)` sets the number in each direction. If not using

EdgyInt, then must be odd so that there is/are centre-patch micro-grid point/planes in each patch.

- `'nEdge'` (not yet implemented), *optional*, default=1, for each patch, the number of 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. If false or omitted, then interpolate from centre-patch planes.

- `'nEnsem'`, *optional-experimental*, default one, but if more, then an ensemble over this number of realisations.

- `'hetCoeffs'`, *optional*, default empty. Supply a 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 (or $n_c = 6$ for the diffusivity tensor). The coefficients are to be the same for each and every patch. However, macroscale variations are catered for by the $n_c$ coefficients being $n_c$ parameters in some macroscale formula.

  - If `nEnsem` $= 1$, then the array of coefficients is just tiled across the patch size to fill up each patch, starting from the $(1, 1, 1)$-point in each patch.

  - If `nEnsem` $> 1$ (value immaterial), then reset `nEnsem` $:= m_x \cdot m_y \cdot m_z$ and construct an ensemble of all $m_x \cdot m_y \cdot m_z$ phase-shifts of the coefficients. In this scenario, the inter-patch coupling couples different members in the ensemble. When `EdgyInt` is true, and when the coefficients are diffusivities/elasticities in $x, y, z$-directions, respectively, then this coupling cunningly preserves symmetry.

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

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

**Output** The struct `patches` is created and set with the following components. If no output variable is provided for `patches`, then make the struct available as a global variable.[7]

```
if nargout==0, global patches, end
```

- `.fun` is the name of the user's function `fun(t,u,patches)` or `fun(t,u)` 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/front and left/bottom/back faces (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 $x_{iI}$ of the microscale grid points in every patch.
- `.y` (8D) is $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the regular spatial locations $y_{jJ}$ of the microscale grid points in every patch.
- `.z` (8D) is $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the regular spatial locations $z_{kK}$ of the microscale grid points in every patch.
- `.ratio` $1 \times 3$, are the size ratios of every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.
- `.le`, `.ri`, `.bo`, `.to`, `.ba`, `.fr` determine inter-patch coupling of members in an ensemble. Each a column vector of length `nEnsem`.
- `.cs` either
  - `[]` 0D, or
  - if `nEnsem` $= 1$, (`nSubP(1)`$-1$)$\times$(`nSubP(2)`$-1$)$\times$(`nSubP(3)`$-1$)$\times n_c$ 4D array of microscale heterogeneous coefficients, or
  - if `nEnsem` $> 1$, (`nSubP(1)` $-$ 1) $\times$ (`nSubP(2)` $-$ 1) $\times$ (`nSubP(3)` $-$ 1) $\times n_c \times m_x m_y m_z$ 5D array of $m_x m_y m_z$ ensemble of phase-shifts of the microscale heterogeneous coefficients.
- `.parallel`, logical: true if patches are distributed over multiple CPUs/cores for the Parallel Computing Toolbox, otherwise false (the default is to activate the *local* pool).
- `.codist`, *optional*, describes the particular parallel distribution of arrays over the active parallel pool.

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

### 3.11.1 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 $\leftrightarrow$ patchSys3 $\leftrightarrow$ user's PDE
3. process results

Set random heterogeneous coefficients of period two in each of the three directions. Crudely normalise by the harmonic mean so the 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.

```
global patches
patches = 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 3.13 shows. This wave propagates diagonally across space.

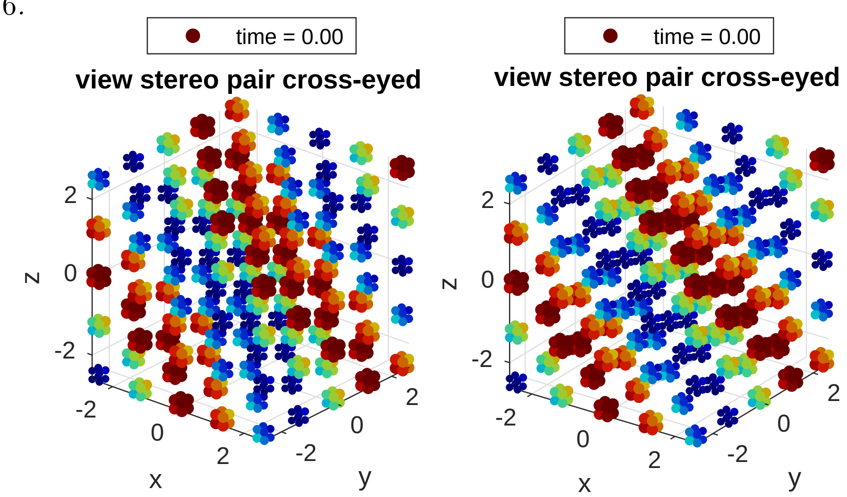
```
u0 = 0.5+0.5*sin(patches.x+patches.y+patches.z);
v0 =    -0.5*cos(patches.x+patches.y+patches.z)*sqrt(3);
uv0 = cat(4,u0,v0);
```

Integrate in time to $t = 6$ using standard functions. In Matlab `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is much quicker (Maclean et al. 2020, Fig. 4).

```
disp('Simulate heterogeneous wave u_tt=div[C*grad(u)]')
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys3,linspace(0,6),uv0(:));
else %disp('octave version is very slow for me')
    lsode_options('absolute tolerance',1e-4);
    lsode_options('relative tolerance',1e-4);
    [ts,us] = odeOcts(@patchSys3,[0 1 2],uv0(:));
end
```

Animate the computed simulation to end with Figure 3.14. Use `patchEdgeInt3` to obtain patch-face values (but not edge nor corner values, and even if not drawn) in order to most easily reconstruct the array data structure.

*Figure 3.13: initial field $u(x, y, z, t)$ at time $t = 0$ of the patch scheme applied to a heterogeneous wave* PDE*: Figure 3.14 plots the computed field at time $t = 6$.*




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

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

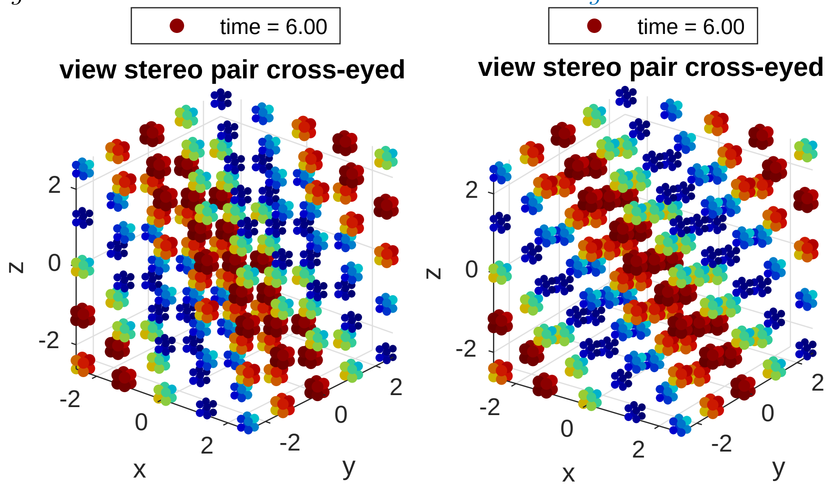
In the scatter plot, these functions `pix()` and `col()` map the $u$-data values to the size of the dots and to the colour of the dots, respectively.

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

Loop to plot at each and every time step.

```
for i = 1:length(ts)
  uv = patchEdgeInt3(us(i,:));
  u = uv(:,:,:,1,:);
  for p=1:2
    subplot(1,2,p)
    if (i==1)| exist('OCTAVE_VERSION','builtin')
      scat(p) = scatter3(xs(j),ys(j),zs(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')
    end % in matlab just update values
```

*Figure 3.14: 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 3.13.*




```
    set(scat(p),'CData',col(u(j)) ...
        ,'SizeData',pix((8+xs(j)-ys(j)+zs(j))/6+0*u(j)));
    legend(['time = ' num2str(ts(i),'%4.2f')],'Location','north')
  end
```

Optionally save the initial condition to graphic file for Figure 3.10, 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
```

### 3.11.2 `heteroWave3()`: heterogeneous Waves

This function codes the lattice heterogeneous waves inside the patches. The wave PDE is

$$u_t = v, \quad v_t = \vec\nabla(C\vec\nabla \cdot u)$$

for diagonal matrix $C$ which has microscale variations. For 8D input arrays `u`, `x`, `y`, and `z` (via edge-value interpolation of `patchSys3`, Section 3.12), computes the time derivative at each point in the interior of a patch, output in `ut`. The three 3D array of heterogeneous coefficients, $c^x_{ijk}$, $c^y_{ijk}$ and $c^z_{ijk}$, have previously been stored in `patches.cs` (4D).

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

```
function ut = heteroWave3(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps, and interior point indices.

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

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
  ut = nan+u;  % preallocate output array
  ut(i,j,k,1,:) = u(i,j,k,2,:);
  ut(i,j,k,2,:) ...
  =diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,1,:),1),1)/dx^2 ...
  +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,1,:),1,2),1,2)/dy^2 ...
  +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,1,:),1,3),1,3)/dz^2;
end% function
```

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

To simulate in time with 3D spatial patches we often need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth enough* so that the patch centre-values are sensible macroscale variables, and patch edge-values are determined by macroscale interpolation of the patch-centre or edge values. Nonetheless, microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (Section 3.11) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

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

### Input

- `u` is a vector/array 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)` spatial grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches3()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,patches)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` $\times$ `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, although 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 $x_i$ of the microscale $(i,j,k)$-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(2)` $\times 1$ array of the spatial locations $y_j$ of the microscale $(i,j,k)$-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.z` is similarly $1 \times 1 \times$ `nSubP(3)` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch(3)` array of the spatial locations $z_k$ of the microscale $(i,j,k)$-grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.

### Output

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

## 3.13 `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-plane values (Roberts et al. 2014, Bunder et al. 2019), or patch next-to-face values which appears better (Bunder et al. 2020). This function is primarily used by `patchSys3()` but is also useful for user graphics. [8]

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

```
function u = patchEdgeInt3(u,patches)
if nargin<2, global patches, end
```

### Input

- `u` is a vector/array of length `prod(nSubP)`·`nVars`·`nEnsem`·`prod(nPatch)` where there are `nVars` · `nEnsem` field values at each of the points in the `nSubP1` · `nSubP2` · `nSubP3` · `nPatch1` · `nPatch2` · `nPatch3` multiscale spatial grid on the `nPatch1` · `nPatch2` · `nPatch3` array of patches.
- `patches` a struct set by `configPatches3()` which includes the following information.
  - `.x` is `nSubP1` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch1` $\times 1 \times 1$ array of the spatial locations $x_{iI}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.y` is similarly $1 \times$ `nSubP2` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch2` $\times 1$ array of the spatial locations $y_{jJ}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.z` is similarly $1 \times 1 \times$ `nSubP3` $\times 1 \times 1 \times 1 \times 1 \times$ `nPatch3` array of the spatial locations $z_{kK}$ of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
  - `.ordCC` is order of interpolation, currently only $\{0, 2, 4, \ldots\}$
  - `.stag` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation. Currently must be zero.
  - `.Cwtsr` and `.Cwtsl` are the coupling coefficients for finite width interpolation in each of the $x, y, z$-directions.
  - `.EdgyInt` true/false is true for interpolating patch-face values from opposite next-to-face values (often preserves symmetry).

[8] Script `patchEdgeInt3test.m` verifies this code.

- `.nEnsem` the number of realisations in the ensemble.
- `.parallel` whether serial or parallel.

**Output**

- `u` is 8D array, `nSubP1` · `nSubP2` · `nSubP3` · `nVars` · `nEnsem` · `nPatch1` · `nPatch2` · `nPatch3`, of the fields with face values set by interpolation (edge and corner vales set to `NaN`).

## 3.14 `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 ...
    ,0, 0.3, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr );
```

### 3.14.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 3.15.

```
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.

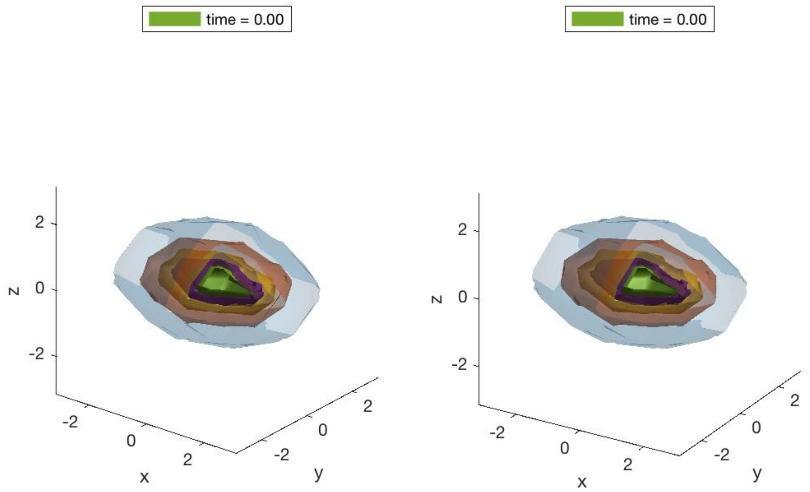
```
if ~exist('OCTAVE_VERSION','builtin')
    [ts,us] = ode23(@patchSys3, 0.3*linspace(0,1,50).^2, u0(:));
else % octave version
    [ts,us] = odeOcts(@patchSys3, 0.3*linspace(0,1).^2, u0(:));
end
```

**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.

*Figure 3.15: 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 3.16*] *plots the isosurfaces of the computed field at time $t = 0.3$.*




```
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 of isosurfaces')
     pause
else pause(0.05)
end
```

Finish the animation loop, and optionally output the isosurfaces of the final field, Figure 3.16.

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

### 3.14.2 Compute Jacobian and its spectrum

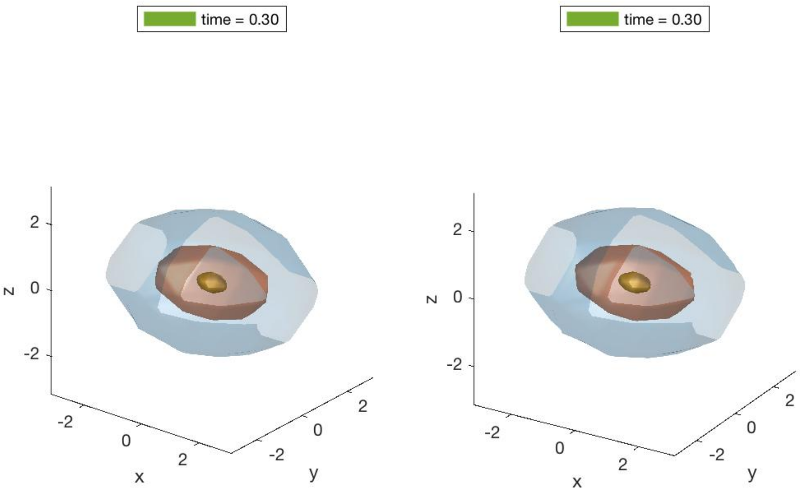
Let's explore the Jacobian dynamics for a range of orders of interpolation, all for the same random patch design and heterogeneity. Except here use a small ratio as we do not plot and then the scale separation is clearest.

```
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;
```

*Figure 3.16: 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.*




```
    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 = patchSys3(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
```

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

This function codes the lattice heterogeneous diffusion inside the patches. For 8D input array `u` (via edge-value interpolation of `patchEdgeInt3`, such as by `patchSys3`, Section 3.12), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The three 3D array of diffusivities, $c^x_{ijk}$, $c^y_{ijk}$ and $c^z_{ijk}$, have previously been stored in `patches.cs` (4+D).

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

```
function ut = heteroDiff3(t,u,patches)
  if nargin<3, global patches, end
```

Microscale space-steps. Q: is using `i,j,k` slower than `2:end-1`??

```
  dx = diff(patches.x(2:3));  % x micro-scale step
  dy = diff(patches.y(2:3));  % y micro-scale step
```

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

Reserve storage and then assign interior patch values to the heterogeneous diffusion time derivatives. Using `nan+u` appears quicker than `nan(size(u),patches.codist)`

```
  ut = nan+u; % reserve storage
  ut(i,j,k,:,:,:,:,:) ...
  = diff(patches.cs(:,j,k,1,:).*diff(u(:,j,k,:,:,:,:,:),1),1)/dx^2 ...
   +diff(patches.cs(i,:,k,2,:).*diff(u(i,:,k,:,:,:,:,:),1,2),1,2)/dy^2 ...
   +diff(patches.cs(i,j,:,3,:).*diff(u(i,j,:,:,:,:,:,:),1,3),1,3)/dz^2;
end% function
```

Fin.

# 4 Matlab parallel computation of the patch scheme

**Chapter contents**



For large-scale simulations, we here assume you have a compute cluster with many independent computer processors linked by a high-speed network. The functions we provide in our toolbox aim to distribute computations in parallel across the cluster. Matlab*'s Parallel Computing Toolbox* empowers a reasonably straightforward way to implement this parallelisation.[1] The reason is that the patch scheme (Chapter 3) has a clear domain decomposition of assigning relatively few patches to each processor.

[1] This parallelisation is not written for, nor tested for, Octave.

The examples listed herein are all *Proof of Principle*: as coded they are all small enough that non-parallel execution is here much quicker that the parallel execution. One needs significantly larger and/or more detailed problems than these examples before parallel execution is effective.

As in all parallel cluster computing, interprocessor communication time all to often dominates. It is important to reduce communication as much as possible compared to computation. Consequently, parallel computing is only effective when there is a very large amount of microscale computation done on each processor per communication—all of the examples listed herein are quite small and so the parallel computation of these is much slower than serial computation. We guesstimate that the microscale code may need, per time-step, of the order of many millions of operations per processor in order for the parallelisation to be useful.

To help minimise communication in time-dependent problems we have drafted a special integrator `RK2mesoPatch`, Section 4.4, that communicates between patches only on a meso-time (Bunder et al. 2016).

## 4.1 `chanDispSpmd`: simulation of a 1D shear dispersion via simulation on small patches across a channel

*Section contents*



Simulate 1D shear dispersion along long thin channel, dispersion that is emergent from micro-scale dynamics in 2D space. Use 1D patches as a Proof of Principle example of parallel computing with `spmd`. In this shear dispersion, although the micro-scale diffusivities are one-ish, the shear causes an effective longitudinal 'diffusivity' of the order of $\mathrm{Pe}^2$—which is typically much larger than the micro-scale diffusivity (Taylor 1953, e.g.).

The spatial domain is the channel (large) $L$-periodic in $x$ and $|y| < 1$. Seek to predict a concentration field $c(x, y, t)$ satisfying the linear advection-diffusion PDE

$$\frac{\partial c}{\partial t} = -\operatorname{Pe} u(y)\frac{\partial c}{\partial x} + \frac{\partial}{\partial x}\Big[\kappa_x(y)\frac{\partial c}{\partial x}\Big] + \frac{\partial}{\partial y}\Big[\kappa_y(y)\frac{\partial c}{\partial y}\Big]. \tag{4.1}$$

where Pe denotes a Peclet number, parabolic advection velocity $u(y) = \frac{3}{2}(1 - y^2)$ with noise, and parabolic diffusivity $\kappa_x(y) = \kappa_y(y) = (1 - y^2)$ with noise. The noise is to be multiplicative and log-normal to ensure advection and diffusion are all positive, and to be periodic in $x$.

For a microscale computation we discretise in space with $x$-spacing $\delta x$, and $n_y$ points over $|y| < 1$ with spacing $\delta y := 2/n_y$ at $y_j := -1 + (j - \frac{1}{2})\delta y$, $j = 1 : n_y$. Our microscale discretisation of PDE (4.1) is then

$$\begin{aligned}\frac{\partial c_{ij}}{\partial t} &= -\operatorname{Pe} u(y_j)\frac{c_{i+1,j} - c_{i-1,j}}{2\delta x} + \frac{d_{i,j+1/2} - d_{i,j-1/2}}{\delta y} + \frac{D_{i+1/2,j} - D_{i-1/2,j}}{\delta x}\,, \\ d_{ij} &:= \kappa_y(y_j)\frac{c_{i,j+1/2} - c_{i,j-1/2}}{\delta y}\,, \quad D_{ij} := \kappa_x(y_j)\frac{c_{i+1/2,j} - c_{i-1/2,j}}{\delta x}\,. \end{aligned} \tag{4.2}$$

These are coded in Section 4.1.4 for the computation.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive interprocessor communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 1
```

The micro-scale PDE is evaluated at positions $y_j$ across the channel, $|y| < 1$. The even indexed points are the collocation points for the PDE, whereas the odd indexed points are the half-grid points for specification of $y$-diffusivities.

```
ny = 7
y = linspace(-1,1,2*ny+1);
yj = y(2:2:end);
```

Set micro-scale advection (array 1) and diffusivity (array 2) with (roughly) parabolic shape (Watt & Roberts 1995, MacKenzie & Roberts 2003, e.g.). Here modify the parabola by a heterogeneous log-normal factor with specified period along the channel: modify the strength of the heterogeneity by the coefficient of `randn` from zero to perhaps one: coefficient 0.3 appears a good moderate value. Remember that `configPatches1` reshapes `cHetr` to 2D.

```
mPeriod = 4
cHetr = shiftdim([3/2 1],-1).*(1-y.^2) ...
        .*exp(0.3*randn([mPeriod 2*ny+1 2]));
```

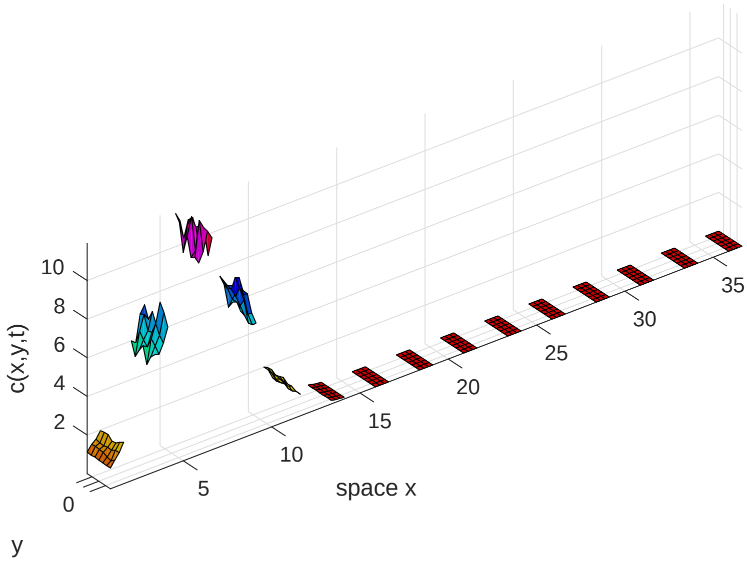
Configure the patch scheme with some arbitrary choices of domain, patches, size ratios. Choose some random order of interpolation to see the alternatives. Set `patches` information to be global so the info can be used for Cases 1–2 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
if theCase<=2, global patches, end
nPatch=15
nSubP=2+mPeriod
ratio=0.2+0.2*(theCase<4)
Len=nPatch/ratio
ordCC=2*randi([0 3])
disp('**** Setting configPatches1')
patches = configPatches1(@chanDispMicro, [0 Len], nan ...
    , nPatch, ordCC, ratio, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

When using parallel then additional parameters to `patches` should be set within a `spmd` block (because `patches` is a co-distributed structure).

```
Peclet = 10
if theCase==1, patches.Pe = Peclet;
else     spmd, patches.Pe = Peclet; end
end
```

*Figure 4.1: initial field $u(x, y, 0)$ of the patch scheme applied to a heterogeneous advection-diffusion* PDE. *Figure 4.2 plots the roughly smooth field values at time $t = 4$. In this example the patches are relatively large, ratio 0.4, for visibility.*




### 4.1.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 4.1.

```
disp('**** Set initial condition and test dc0dt =')
if theCase==1
```

Without parallel processing, invoke the usual operations.

```
    c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
    c0 = c0.*(1+0.2*rand(size(c0)));
    dc0dt = patchSys1(0,c0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes use `patches.codist` to explicitly code how to distribute new arrays over the cpus. Now `patchSys1` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we *must* pass it explicitly as a parameter to `patchSys1`.

```
else, spmd
    c0 = 10*exp(-(ratio*patches.x-2.5).^2/2) +0*yj;
    c0 = c0.*(1+0.2*rand(size(c0),patches.codist));
    dc0dt = patchSys1(0,c0,patches)
    end%spmd
end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up to 0.3, need micro-steps less than 0.005; with noise 1, need micro-steps less than 0.0015.

```
warning('Integrating system in time, wait patiently')
ts=4*linspace(0,1);
```

Go to the selected case.

```
switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys1` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
case 1
%     [cs,uerrs] = RK2mesoPatch(ts,c0);
    [ts,cs] = ode23(@patchSys1,ts,c0(:));
    cs=reshape(cs,[length(ts) size(c0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
case 2
    cs = RK2mesoPatch(ts,c0);
    cs = cs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
case 3,spmd
    cs = RK2mesoPatch(ts,c0,[],patches);
    end%spmd
    cs = cs{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 4.3.3) to compute each burst of the micro-scale simulation. For a Peclet number of ten, the macro-scale time-step needs to be less than about 0.5 (which here is very little projection)—presumably the mean advection in a macro-step needs to be less than about the patch spacing. The function `microBurst()` here interfaces to `aBurst()` (Section 4.1.3) in order to provide shaped initial states, and to provide the patch information.

```
case 4
    microBurst = @(tb0,xb0,bT) ...
        aBurst(tb0 ,reshape(xb0,size(c0)) ,patches);
    ts = 0:0.7:5
    cs = PIRK2(microBurst,ts,gather(c0(:)));
    cs = reshape(cs,[length(ts) size(c0)]);
```

End the four cases.

```
end%switch theCase
```

### 4.1.2 Plot the solution

Optionally set to save some plots to file.

```
if 0, global OurCf2eps, OurCf2eps=true, end
```

**Animate the computed solution field over time**

```
figure(1), clf, colormap(0.8*hsv)
```

First get the $x$-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
if theCase==1, x = patches.x;
else, spmd
    x = gather( patches.x );
    end%spmd
    x = x{1};
end
x([1 end],:,:,:) = nan;
```

For every time step draw the concentration values as a set of surfaces on 2D patches, with a short pause to display animation.

```
nTimes = length(ts)
for l = 1:nTimes
```

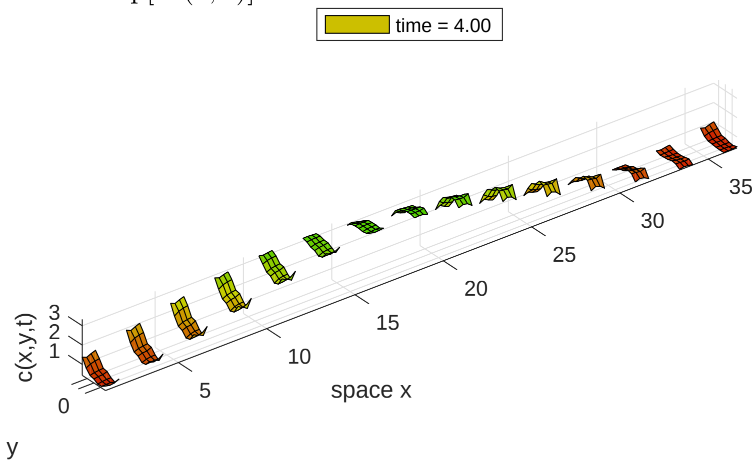
At each time, squeeze sub-patch data into a 3D array, permute to get all the $x$-variation in the first two dimensions, and reshape into $x$-variation for each and every ($y$).

```
  c = reshape( permute( squeeze( ...
      cs(l,:,:,:,:) ) ,[1 3 2]) ,numel(x),ny);
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space.

```
  if l==1
    hp = surf(x(:),yj,c');
    axis([0 Len -1 1 0 max(c(:))])
    axis equal
    xlabel('space x'), ylabel('y'); zlabel('c(x,y,t)')
    ifOurCf2eps([mfilename 't0'])
    legend(['time = ' num2str(ts(l),'%4.2f')] ...
        ,'Location','north')
    disp('**** pausing, press blank to animate')
    pause
 else
    hp.ZData = c';
    legend(['time = ' num2str(ts(l),'%4.2f')])
```

*Figure 4.2: final field* $c(x, y, 4)$ *of the patch scheme applied to a heterogeneous advection-diffusion* PDE (4.1) *with heterogeneous factor log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
    pause(0.1)
 end
```

Finish the animation loop, and optionally save the final plot to file, Figure 4.2.

```
end%for over time
ifOurCf2eps([mfilename 'tFin'])
```

**Macro-scale view** Plot a macro-scale mesh of the predictions: at each of a selection of times, for every patch, plot the patch-mean value at the mean-$x$.

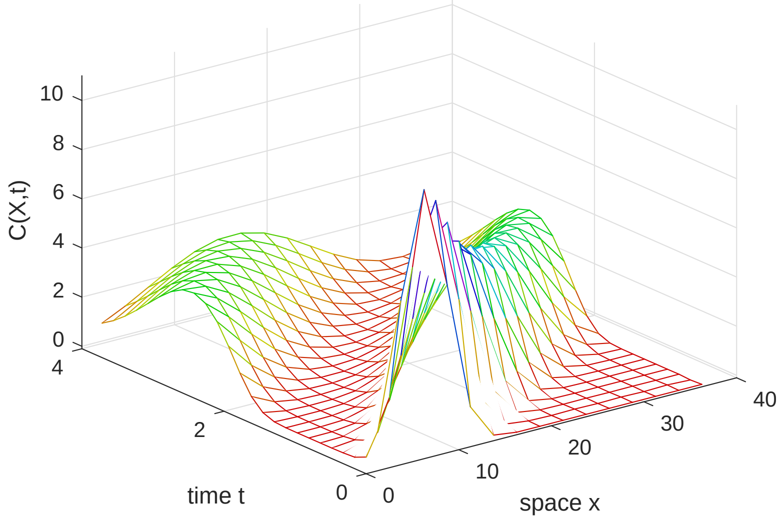
```
figure(2), clf, colormap(0.8*hsv)
X = squeeze(mean(x(2:end-1,:,:,:)));
C = squeeze(mean(mean(cs(:,2:end-1,:,:,:),2),3));
j = 1:ceil(nTimes/30):nTimes;
mesh(X,ts(j),C(j,:));
xlabel('space x'),ylabel('time t'),zlabel('C(X,t)')
zlim([-0.1 11])
ifOurCf2eps([mfilename 'Macro'])
```

### 4.1.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.2. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0033. With macro-step 0.5, these parameters usually give stable projective integration.

```
function [tbs,xbs] = aBurst(tb0,xb0,patches)
    normx=max(abs(xb0(:)));
    disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
    assert(normx<20,'solution exploding')
    tbs = tb0+(0:0.033:0.2);
```

*Figure 4.3: macro-scale view of heterogeneous advection-diffusion* PDE *along a (periodic) channel obtained via the patch scheme.*




```
    spmd
      xb0 = codistributed(xb0,patches.codist);
      xbs = RK2mesoPatch(tbs,xb0,[],patches);
    end%spmd
    xbs=reshape(xbs{1},length(tbs),[]);
end%function
```

Fin.

#### 4.1.4 `chanDispMicro()`: heterogeneous 2D advection-diffusion in a long thin channel

This function codes the lattice heterogeneous diffusion inside the patches. For 4D input arrays of concentration `c` and spatial lattice `x` (via edge-value interpolation of `patchSys1`, Section 3.2), computes the time derivative (4.2) at each point in the interior of a patch, output in `ct`. The heterogeneous advections and diffusivities, $u_i(y_j)$ and $\kappa_i(y_{j+1/2})$, have previously been merged and stored in the one array `patches.cs` (2D).

```
function ct = chanDispMicro(t,c,p)
  [nx,ny,~,~]=size(c); % micro-grid points in patches
  ix = 2:nx-1;         % x interior points in a patch
  dx = diff(p.x(2:3)); % x space step
  dy = 2/ny;           % y space step
  ct = nan+c;          % preallocate output array
  pcs = reshape(p.cs,nx-1,[],2);
```

Compute the cross-channel flux using 'ghost' nodes at channel boundaries, so that the flux is zero at $y = \pm 1$ either because the boundary values are replicated so the differences are zero, or because the diffusivities in `cs` are zero at the channel boundaries.

```
  ydif = pcs(ix,1:2:end,2) ...
         .*(c(ix,[1:end end],:,:)-c(ix,[1 1:end],:,:))/dy;
```

Now evaluate advection-diffusion time derivative (4.2). Could use upwind advection and no longitudinal diffusion, or, as here, centred advection and diffusion.

```
  ct(ix,:,:,:) = (ydif(:,2:end,:,:)-ydif(:,1:end-1,:,:))/dy ...
  + diff(pcs(:,2:2:end,2).*diff(c))/dx^2 ...
  - p.Pe*pcs(ix,2:2:end,1).*(c(ix+1,:,:,:)-c(ix-1,:,:,:))/(2*dx);
end% function
```

## 4.2 `rotFilmSpmd`: simulation of a 2D shallow water flow on a rotating heterogeneous substrate

*Section contents*



As an example application, consider the flow of a shallow layer of fluid on a solid flat rotating substrate, such as in spin coating (Wilson et al. 2000, Oron et al. 1997, §II.K, e.g.) or large-scale shallow water waves (Dellar & Salmon 2005, Hereman 2009, e.g.). Let $\vec{x} = (x, y)$ parametrise location on the rotating substrate, and let the fluid layer have thickness $h(\vec{x}, t)$ and move with depth-averaged horizontal velocity $\vec{v}(\vec{x}, t) = (u, v)$. We take as given (with its simplified physics) that the (non-dimensional) governing set of PDEs is the nonlinear system (Bunder & Roberts 2018, eq. (1), e.g.)

$$\frac{\partial h}{\partial t} = -\nabla \cdot (h\vec{v}), \tag{4.3a}$$

$$\frac{\partial \vec{v}}{\partial t} = \begin{bmatrix} -b & f \\ -f & -b \end{bmatrix} \vec{v} - (\vec{v} \cdot \nabla)\vec{v} - g\nabla h + \vec{\nabla} \cdot (\nu \vec{\nabla} \vec{v}) , \tag{4.3b}$$

where $b(\vec{x})$ represents heterogeneous 'bed' drag, $f$ is the Coriolis coefficient, $g$ is the acceleration due to gravity, $\nu(\vec{x})$ is a heterogeneous 'kinematic viscosity', and we neglect surface tension.

The aim is to simulate the macroscale dynamics which (for constant $b$) is approximately that of the nonlinear diffusion $\partial h/\partial t \approx \frac{gb}{b^2+f^2} \vec{\nabla} \cdot (h\vec{\nabla} h)$ (Bunder & Roberts 2018, eq. (2)). But there is no known algebraic closure for the macroscale in the case of heterogeneous $b(\vec{x})$ and $\nu(\vec{x})$, nonetheless the patch scheme automatically predicts a sensible macroscale for such heterogeneous dynamics (Figure 4.5).

For the microscale computation, Section 4.2.4 discretises the PDEs (4.3) in space with $x, y$-spacing $\delta x, \delta y$.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` illustrates that `RK2mesoPatch` invokes `spmd` computation if parallel has been configured.
- `theCase=3` shows how users explicitly invoke `spmd`-blocks around the time integration.

- theCase=4 invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within spmd-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 1
```

Set micro-scale bed drag (array 1) and diffusivity (arrays 2–3) to be a heterogeneous log-normal factor with specified period: modify the strength of the heterogeneity by the coefficient of randn from zero to perhaps one: coefficient 0.3 appears a good moderate value.

```
mPeriod = 5
bnu = shiftdim([1 0.5 0.5],-1) ...
      .*exp(0.3*randn([mPeriod mPeriod 3]));
```

Configure the patch scheme with these choices of domain, patches, size ratios—here each patch is square in space. In Cases 1–2, set patches information to be global so the info can be used without being explicitly passed as arguments.

```
if theCase<=2, global patches, end
```

In Case 4, double the size of the domain and use more separated patches accordingly, to maintain the spatial microscale grid spacing to be 0.055. Here use fourth order edge-based coupling between patches. Choose the parallel option if not Case 1, which invokes spmd-block internally, so that field variables become *distributed* across cpus.

```
nSubP = 2+mPeriod
nPatch = 9
ratio = 0.2+0.2*(theCase<4)
Len = 2*pi*(1+(theCase==4))
disp('**** Setting configPatches2')
patches = configPatches2(@rotFilmMicro, [0 Len], nan ...
    , nPatch, 4, ratio, nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',bnu ,'parallel',(theCase>1) );
```

When using parallel, any additional parameters to patches, such as physical parameters for the microcode, must be set within a spmd block (because patches is a co-distributed structure). Here set frequency of substrate rotation, and strength of gravity.

```
f = 5,  g = 1
if theCase==1, patches.f = f; patches.g = g;
else     spmd, patches.f = f; patches.g = g; end
end
```

### 4.2.1 Simulate heterogeneous advection-diffusion

Set initial conditions of a simulation as shown in Figure 4.4. Here the initial condition is a (periodic) quasi-Gaussian in $h$ and zero velocity $\vec{v}$, with additive random perturbations.

```
disp('**** Set initial condition and test dhuv0dt =')
if theCase==1
```

When not parallel processing, invoke the usual operations. Here add a random noise to the velocity field, but keep $h(x, y, 0)$ smooth as shown by Figure 4.4. The `shiftdim(...,-1)` moves the given row-vector of coefficients into the third dimension to become coefficients of the fields $(h, u, v)$, respectively.

```
    huv0 = shiftdim([0.5 0 0],-1) ...
       .*exp(-cos(patches.x)/2-cos(patches.y));
    huv0 = huv0+0.1*shiftdim([0 1 1],-1).*rand(size(huv0));
    dhuv0dt = patchSys2(0,huv0);
```

With parallel, we must use an `spmd`-block for computations: there is no difference in Cases 2–4 here. Also, we must sometimes explicitly tell functions how to distribute some initial condition arrays over the cpus. Now `patchSys2` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside an `spmd`-block we *must* pass `patches` explicitly as a parameter to `patchSys2`.

```
else, spmd
    huv0 = shiftdim([0.5 0 0],-1) ...
       .*exp(-cos(patches.x)/2-cos(patches.y));
    huv0 = huv0+0.1*rand(size(huv0),patches.codist);
    dhuv0dt = patchSys2(0,huv0,patches)
    end%spmd
end%if theCase
```

Integrate in time, either via the automatic `ode23` or via `RK2mesoPatch` which reduces communication between patches. By default, `RK2mesoPatch` does ten micro-steps for each specified meso-step in `ts`. For stability: with noise up to 0.3, need micro-steps less than 0.0003; with noise 1, need micro-steps less than 0.0001.

```
warning('Integrating system in time, wait a minute')
ts=0:0.003:0.3;
```

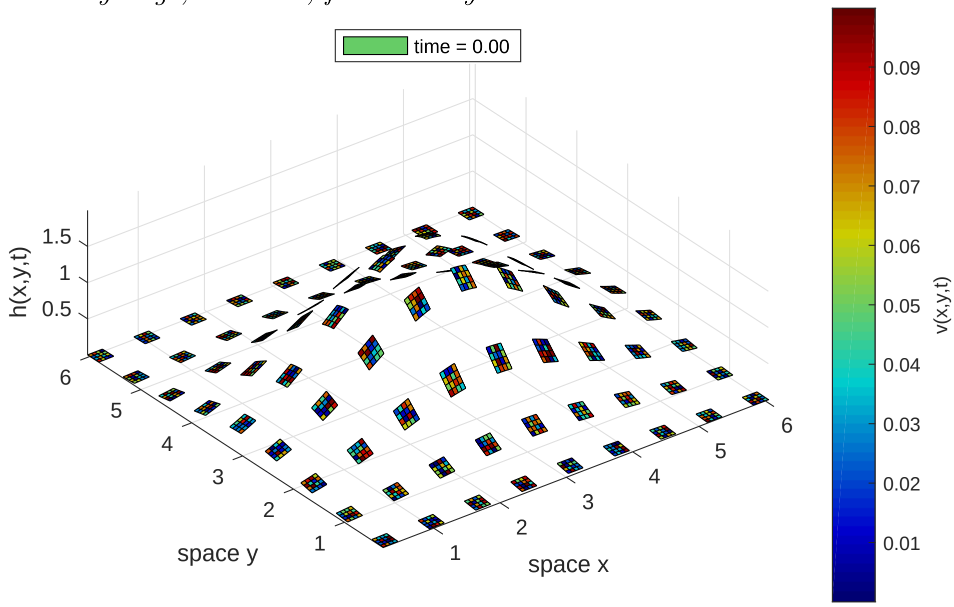
Go to the selected case.

```
switch theCase
```

1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys2` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—namely that of the initial condition.

```
case 1
%     tic,[huvs,uerrs] = RK2mesoPatch(ts,huv0);toc
     [ts,huvs] = ode23(@patchSys2,[0 4],huv0(:));
     huvs=reshape(huvs,[length(ts) size(huv0)]);
```

*Figure 4.4: initial field $h(x, y, 0)$ of the patch scheme applied to the heterogeneous, shallow water, rotating substrate,* PDE (4.3)*. The micro-scale sub-patch colour displays the initial $y$-direction velocity field $v(x, y, 0)$. Figure 4.5 plots the roughly smooth field values at time $t = 6$. In this example the patches are relatively large, ratio* 0.4*, for visibility.*




2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
case 2
    huvs = RK2mesoPatch(ts,huv0);
    huvs = huvs{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
case 3,spmd
    huvs = RK2mesoPatch(ts,huv0,[],patches);
    end%spmd
    huvs = huvs{1};
```

4. In this fourth case, use Projective Integration (PI). Currently the PI is done serially, with parallel `spmd`-blocks only invoked inside function `aBurst()` (Section 4.2.3) to compute each burst of the micro-scale simulation. The macro-scale time-step needs to be less than about 0.1 (which here is not much projection). The function `microBurst()` interfaces to `aBurst()` (Section 4.2.3) in order to provide shaped initial states, and to provide the patch information.

```
case 4
    microBurst = @(tb0,xb0,bT) ...
```

```
            aBurst(tb0 ,reshape(xb0,size(huv0)) ,patches);
        ts = 0:0.1:1
        huvs = PIRK2(microBurst,ts,gather(huv0(:)));
        huvs = reshape(huvs,[length(ts) size(huv0)]);
```

End the four cases.

```
end%switch theCase
```

### 4.2.2 Plot the solution

Optionally set to save some plots to file.

```
if 0, global OurCf2eps, OurCf2eps=true, end
```

**Animate the computed solution field over time**

```
figure(1), clf, colormap(0.8*jet)
```

First get the $x$-coordinates and omit the patch-edge values from the plot (because they are not here interpolated).

```
if theCase==1, x = patches.x;
               y = patches.y;
else, spmd
    x = gather( patches.x );
    y = gather( patches.y );
    end%spmd
    x = x{1}; y = y{1};
end
x([1 end],:,:,:,:,:) = nan;
y(:,[1 end],:,:,:,:) = nan;
```

Draw the field values as a patchy surface evolving over 100–200 time steps.

```
nTimes = length(ts)
for l = 1:ceil(nTimes/200):nTimes
```

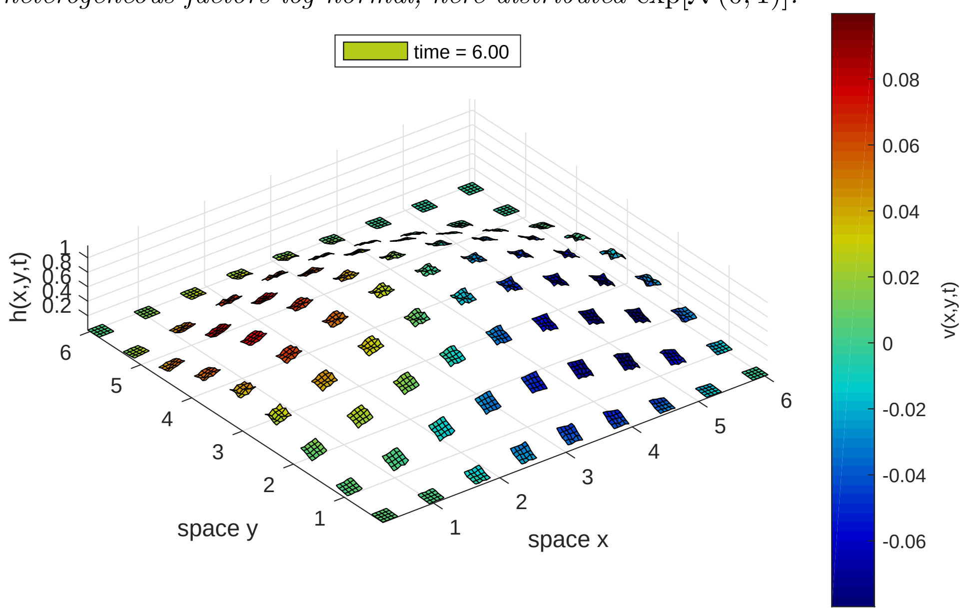
At each time, squeeze sub-patch data fields into three 4D arrays, permute to get all the $x/y$-variations in the first/last two dimensions, and and then reshape to 2D.

```
  h = reshape( permute( squeeze( ...
      huvs(l,:,:,1,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
  u = reshape( permute( squeeze( ...
      huvs(l,:,:,2,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
  v = reshape( permute( squeeze( ...
      huvs(l,:,:,3,1,:,:) ) ,[1 3 2 4]) ,numel(x),numel(y));
```

Draw surface of each patch, to show both micro-scale and macro-scale variation in space. Colour the surface according to the velocity $v$ in the $y$-direction.

```
  if l==1
    hp = surf(x(:),y(:),h',v');
```

*Figure 4.5: final field $h(x, y, 6)$, coloured by $v(x, y, 6)$, of the patch scheme applied to the heterogeneous, shallow water, rotating substrate,* PDE (4.3) *with heterogeneous factors log-normal, here distributed* $\exp[\mathcal{N}(0, 1)]$.




```
    axis([0 Len 0 Len 0 max(h(:))])
    c = colorbar; c.Label.String = 'v(x,y,t)';
    legend(['time = ' num2str(ts(l),'%4.2f')] ...
        ,'Location','north')
    axis equal
    xlabel('space x'), ylabel('space y'), zlabel('h(x,y,t)')
    ifOurCf2eps([mfilename 't0'])
    disp('**** pausing, press blank to begin animation')
    pause
  else
    hp.ZData = h'; hp.CData = v';
    legend(['time = ' num2str(ts(l),'%4.2f')])
    pause(0.1)
  end
```

Finish the animation loop, and optionally save the final plot to file, Figure 4.5.

```
end%for over time
ifOurCf2eps([mfilename 'tFin'])
```

### 4.2.3 `microBurst` function for Projective Integration

Projective Integration stability appears to require bursts longer than 0.01. Each burst is done in parallel processing. Here use `RK2mesoPatch` to take take meso-steps, each with default ten micro-steps so the micro-scale step is 0.0003. With macro-step 0.1, these parameters usually give stable projective integration.

```
function [tbs,xbs] = aBurst(tb0,xb0,patches)
    normx=max(abs(xb0(:)));
    disp(['* aBurst t=' num2str(tb0) '  |x|=' num2str(normx)])
    assert(normx<20,'solution exploding')
    tbs = tb0+(0:0.003:0.015);
    spmd
      xb0 = codistributed(xb0,patches.codist);
      xbs = RK2mesoPatch(tbs,xb0,[],patches);
    end%spmd
    xbs=reshape(xbs{1},length(tbs),[]);
end%function
```

Fin.

### 4.2.4 `rotFilmMicro()`: 2D shallow water flow on a rotating heterogeneous substrate

This function codes the heterogeneous shallow water flow (4.3) inside 2D patches. The PDEs are discretised on the multiscale lattice in terms of evolving variables $h_{ijIJ}$, $u_{ijIJ}$ and $v_{ijIJ}$. For 6D input array `huv` (via edge-value interpolation of `patchEdgeInt2()`, Section 3.8), computes the time derivatives (4.3) at each point in the interior of a patch, output in `huvt`. The heterogeneous bed drag and diffusivities, $b_{ij}$ and $\nu_{ij}$, have previously been merged and stored in the array `patches.cs` (2D $\times$ 3): herein `patches` is named `p`.

```
function huvt = rotFilmMicro(t,huv,p)
  [nx,ny,~]=size(huv); % micro-grid points in patches
  i = 2:nx-1;          % x interior points in a patch
  j = 2:ny-1;          % y interior points in a patch
  dx = diff(p.x(2:3)); % x space step
  dy = diff(p.y(2:3)); % y space step
  huvt = nan+huv;      % preallocate output array
```

Set indices of fields in the arrays. Need to store different diffusivity values for the $x, y$-directions as they are evaluated at different points in space.

```
  h=1; u=2; v=3;
  b=1; nux=2; nuy=3;
```

Use a staggered micro-grid so that `h(i,j)` $= h_{ij}$, `u(i,j)` $= u_{i+1/2,j}$, and `v(i,j)` $= v_{i,j+1/2}$. We need the following to interpolate some quantities to other points on the staggered micro-grid. But the first two statements fill-in two needed corner values because they are not (currently) interpolated by `patchEdgeInt2()`.

```
huv(1,ny,u,:,:,:) = huv(2,ny,u,:,:,:)+huv(1,ny-1,u,:,:,:) ...
                   -huv(2,ny-1,u,:,:,:);
huv(nx,1,v,:,:,:) = huv(nx,2,v,:,:,:)+huv(nx-1,1,v,:,:,:) ...
                   -huv(nx-1,2,v,:,:,:);
v4u = (huv(i,j-1,v,:,:,:)+huv(i+1,j,v,:,:,:) ...
      +huv(i,j,v,:,:,:)+huv(i+1,j-1,v,:,:,:))/4;
```

```
u4v = (huv(i,j+1,u,:,:,:)+huv(i-1,j,u,:,:,:) ...
      +huv(i,j,u,:,:,:)+huv(i-1,j+1,u,:,:,:))/4;
h2u = (huv(2:nx,:,h,:,:,:)+huv(1:nx-1,:,h,:,:,:))/2;
h2v = (huv(:,2:ny,h,:,:,:)+huv(:,1:ny-1,h,:,:,:))/2;
```

Evaluate conservation of mass PDE (4.3a) (needing averages of $h$ at half-grid points):

```
  huvt(i,j,h,:,:,:) = ...
    - (h2u(i,j  ,:,:,:,:).*huv(i  ,j,u,:,:,:) ...
      -h2u(i-1,j,:,:,:,:).*huv(i-1,j,u,:,:,:) )/dx ...
    - (h2v(i,j  ,:,:,:,:).*huv(i,j  ,v,:,:,:) ...
      -h2v(i,j-1,:,:,:,:).*huv(i,j-1,v,:,:,:))/dy ;
```

Evaluate the $x$-direction momentum PDE (4.3b) (needing to interpolate component $v$ to $u$-points):

```
  huvt(i,j,u,:,:,:) = ...
    - p.cs(i,j,b).*huv(i,j,u,:,:,:) + p.f.*v4u ...
    - huv(i,j,u,:,:,:).*(huv(i+1,j,u,:,:,:)-huv(i-1,j,u,:,:,:))/(2*dx) ...
    - v4u.*(huv(i,j+1,u,:,:,:)-huv(i,j-1,u,:,:,:))/(2*dy) ...
    - p.g*(huv(i+1,j,h,:,:,:)-huv(i,j,h,:,:,:))/dx ...
    + diff(p.cs(:,j,nux).*diff(huv(:,j,u,:,:,:),[],1),[],1)/dx^2 ...
    + diff(p.cs(i,:,nuy).*diff(huv(i,:,u,:,:,:),[],2),[],2)/dy^2 ;
```

Evaluate the $y$-direction momentum PDE (4.3b) (needing to interpolate component $u$ to $v$-points):

```
  huvt(i,j,v,:,:,:) = ...
    - p.cs(i,j,b).*huv(i,j,v,:,:,:) - p.f.*u4v ...
    - u4v.*(huv(i+1,j,v,:,:,:)-huv(i-1,j,v,:,:,:))/(2*dx) ...
    - huv(i,j,v,:,:,:).*(huv(i,j+1,v,:,:,:)-huv(i,j-1,v,:,:,:))/(2*dy) ...
    - p.g*(huv(i,j+1,h,:,:,:)-huv(i,j,h,:,:,:))/dy ...
    + diff(p.cs(:,j,nux).*diff(huv(:,j,v,:,:,:),[],1),[],1)/dx^2 ...
    + diff(p.cs(i,:,nuy).*diff(huv(i,:,v,:,:,:),[],2),[],2)/dy^2 ;
end% function
```

## 4.3 `homoDiff31spmd`: computational homogenisation of a 1D dispersion via parallel simulation on small 3D patches of heterogeneous diffusion

*Section contents*



Simulate effective dispersion along 1D space on 3D patches of heterogeneous diffusion as a Proof of Principle example of parallel computing with `spmd`. With only one patch in each of the $y, z$-directions, the solution simulated is strictly periodic in $y, z$ with period `ratio`: there are only macro-scale variations in the $x$-direction. The discussion here only addresses issues with `spmd` parallel computing. For discussion on the 3D patch scheme with heterogeneous diffusion, see code and documentation for `homoDiffEdgy3` in Section 3.14.

Choose one of four cases:

- `theCase=1` is corresponding code without parallelisation (in this toy problem it is much the quickest because there is no expensive communication);
- `theCase=2` for minimising coding by a user of `spmd`-blocks;
- `theCase=3` is for users happier to explicitly invoke `spmd`-blocks.
- `theCase=4` invokes projective integration for long-time simulation via short bursts of the micro-computation, bursts done within `spmd`-blocks for parallel computing.

First, clear all to remove any existing globals, old composites, etc—although a parallel pool persists. Then choose the case.

```
clear all
theCase = 1
```

Set micro-scale heterogeneity with various spatial periods in the three directions.

```
mPeriod = [4 3 2] %1+randperm(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—here each patch is a unit cube in space. Choose some random order of interpolation. Set `patches` information to be global so the info can be used for Case 1 without being explicitly passed as arguments. Choose the parallel option if not Case 1, which invokes `spmd`-block internally, so that field variables become *distributed* across cpus.

```
if any(theCase==[1 2]), global patches, end
nSubP=mPeriod+2
nPatch=[9 1 1]
ratio=0.3
Len=nPatch(1)/ratio
ordCC=2*randi([0 3])
disp('**** Setting configPatches3')
patches = configPatches3(@heteroDiff3,[0 Len 0 1 0 1], nan ...
    , nPatch, ordCC, [ratio 1 1], nSubP, 'EdgyInt',true  ...
    ,'hetCoeffs',cHetr ,'parallel',(theCase>1) );
```

### 4.3.1 Simulate heterogeneous diffusion

Set initial conditions of a simulation as shown in Figure 4.6.

```
disp('**** Set initial condition and testing du0dt =')
if theCase==1
```

Without parallel processing, invoke the usual operations.

```
    u0 = exp( -(patches.x-Len/2).^2/Len ...
              -patches.y.^2/2-patches.z.^2 );
    u0 = u0.*(1+0.2*rand(size(u0)));
    du0dt = patchSys3(0,u0);
```

With parallel, must use an `spmd`-block for computations: there is no difference in cases 2–4 here. Also, we must sometimes explicitly code how to distribute some new arrays over the cpus. Now `patchSys3` does not invoke `spmd` so higher level code must, as here. Even if `patches` is global, inside `spmd`-block we must pass it explicitly as a parameter to `patchSys3`.

```
else, spmd
    u0 = exp( -(patches.x-Len/2).^2/Len ...
              -patches.y.^2/2-patches.z.^2/4 );
    u0 = u0.*(1+0.2*rand(size(u0),patches.codist));
    du0dt = patchSys3(0,u0,patches);
    end%spmd
end%if theCase
```

Integrate in time. Use non-uniform time-steps for fun, and to show more of the initial rapid transients.

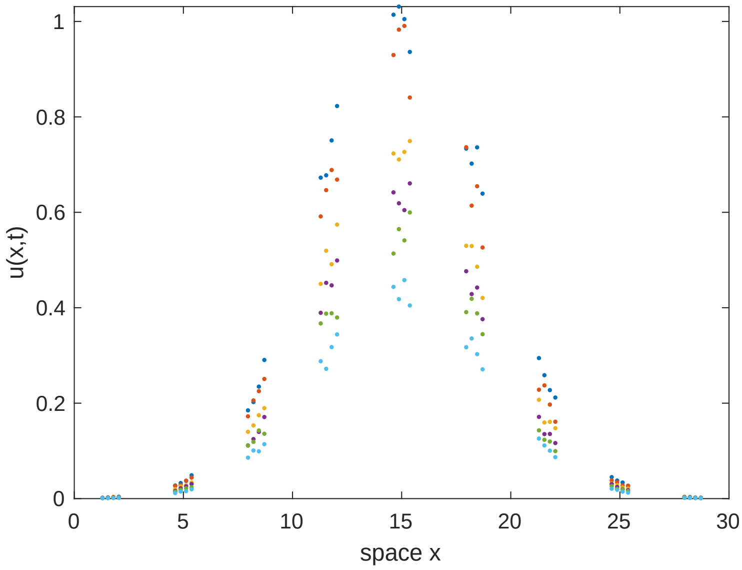
Alternatively, use `RK2mesoPatch` which reduces communication between patches, recalling that, by default, `RK2mesoPatch` does ten micro-steps for each specified step in `ts`. For unit cube patches, need micro-steps less than about 0.004 for stability.

```
warning('Integrating system in time, wait patiently')
ts=0.4*linspace(0,1,21).^2;
```

Go to the selected case.

```
switch theCase
```

*Figure 4.6: initial field $u(x, y, z, 0)$ of the patch scheme applied to a heterogeneous diffusion* PDE*. The vertical spread indicates the extent of the structure in $u$ in the cross-section variables $y, z$. Figure 4.7 plots the nearly smooth field values at time $t = 0.4$.*




1. For non-parallel, we could use `RK2mesoPatch` as indicated below, but instead choose to use standard `ode23` as here `patchSys3` accesses patch information via global `patches`. For post-processing, reshape each and every row of the computed solution to the correct array size—that of the initial condition.

```
case 1
%     [us,uerrs] = RK2mesoPatch(ts,u0);
     [ts,us] = ode23(@patchSys3,ts,u0(:));
     us=reshape(us,[length(ts) size(u0)]);
```

2. In the second case, `RK2mesoPatch` detects a parallel patch code has been requested, but has only one cpu worker, so it auto-initiates an `spmd`-block for the integration. Both this and the next case return *composite* results, so just keep one version of the results.

```
case 2
     us = RK2mesoPatch(ts,u0);
     us = us{1};
```

3. In this third case, a user could merge this explicit `spmd`-block with the previous one that sets the initial conditions.

```
case 3,spmd
     us = RK2mesoPatch(ts,u0,[],patches);
     end%spmd
     us = us{1};
```

4. In this fourth case, use Projective Integration (PI) over long times (`PIRK4` also works). Currently the PI is done serially, with parallel

spmd-blocks only invoked inside function aBurst() (Section 4.3.3) to compute each burst of the micro-scale simulation. A macro-scale time-step of about 3 seems good to resolve the decay of the macro-scale 'homogenised' diffusion. [2] The function microBurst() here interfaces to aBurst() (Section 4.3.3) in order to provide shaped initial states, and to provide the patch information.

```
case 4
    microBurst = @(tb0,xb0,bT) ...
        aBurst(tb0 ,reshape(xb0,size(u0)) ,patches);
    ts = 0:3:51
    us = PIRK2(microBurst,ts,gather(u0(:)));
    us = reshape(us,[length(ts) size(u0)]);
```

End the four cases.

```
end%switch theCase
```

### 4.3.2 Plot the solution

Optionally save some plots to file.

```
if 0, global OurCf2eps, OurCf2eps=true, end
```

Animate the solution field over time. Since the spatial domain is long in $x$ and thin in $y, z$, just plot field values as a function of $x$.

```
figure(1), clf
if theCase==1
    x = reshape( patches.x(2:end-1,:,:,:) ,[],1);
else, spmd
    x = reshape(gather( patches.x(2:end-1,:,:,:) ),[],1);
    end%spmd
    x = x{1};
end
```

For every time step draw the field values as dots and pause for a short display.

```
nTimes = length(ts)
for l = 1:length(ts)
```

At each time, squeeze interior point data into a 4D array, permute to get all the $x$-variation in the first two dimensions, and reshape into $x$-variation for each and every $(y, z)$.

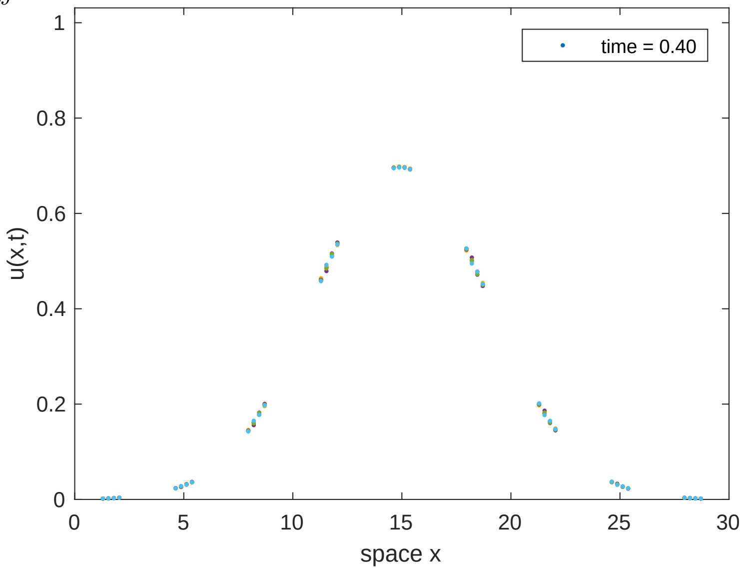
```
  u = reshape( permute( squeeze( ...
      us(l,2:end-1,2:end-1,2:end-1,:) ) ,[1 4 2 3]) ,numel(x),[]);
```

Draw point data to show spread at each cross-section, as well as macro-scale variation in the long space direction.

```
  if l==1
    hp = plot(x,u,'.');
```

[2] Curiously, PIG() appears to suffer unrecoverable instabilities with its variable step size!

*Figure 4.7: final field $u(x, y, z, 0.4)$ of the patch scheme applied to a heterogeneous diffusion* PDE.




```
    axis([0 Len 0 max(u(:))])
    xlabel('space x'), ylabel('u(x,y,z,t)')
    ifOurCf2eps([mfilename 't0'])
    legend(['time = ' num2str(ts(l),'%4.2f')])
    disp('**** pausing, press blank to animate')
    pause
 else
    for p=1:size(u,2), hp(p).YData=u(:,p); end
    legend(['time = ' num2str(ts(l),'%4.2f')])
    pause(0.1)
 end
```

Finish the animation loop, and optionally output the final plot, Figure 4.7.

```
end%for over time
ifOurCf2eps([mfilename 'tFin'])
```

### 4.3.3 `microBurst` function for Projective Integration

Projective Integration stability seems to need bursts longer than 0.2. Here take ten meso-steps, each with default ten micro-steps so the micro-scale step is 0.002. With macro-step 3, these parameters usually give stable projective integration (but not always).

```
function [tbs,xbs] = aBurst(tb0,xb0,patches)
    normx=max(abs(xb0(:)));
    disp(['aBurst t = ' num2str(tb0) '  |x| = ' num2str(normx)])
    assert(normx<10,'solution exploding')
    tbs = tb0+(0:0.02:0.2);
    spmd
      xb0 = codistributed(xb0,patches.codist);
```

```
      xbs = RK2mesoPatch(tbs,xb0,[],patches);
    end%spmd
    xbs=reshape(xbs{1},length(tbs),[]);
end%function
```

Fin.

## 4.4 RK2mesoPatch()

This is a Runge–Kutta, 2nd order, integration of a given deterministic system of ODEs on patches. It invokes meso-time updates of the patch-edge values in order to reduce interpolation costs, and uses a linear variation in edge-values over the meso-time-step (Bunder et al. 2016, case $Q = 2$). This function is aimed primarily for large problems executed on a computer cluster to markedly reduce expensive communication between computers.

If using within projective integration, it appears quite tricky to get all the time-steps chosen appropriately. One has to choose times for: the micro-scale time-step, the meso-time interval between communications, the longer meso-time burst length, and the macro-scale integration time-step.

```
function [xs,errs] = RK2mesoPatch(ts,x0,nMicro,patches)
if nargin<4, global patches, end
```

**Input**

- `patches.fun()` is a function such as `dxdt=fun(t,x,patches)` that computes the right-hand side of the ODE $d\vec{x}/dt = \vec{f}(t,\vec{x})$ where $\vec{x}$ is a vector/array, $t$ is a scalar, and the result $\vec{f}$ is a correspondingly sized vector/array.
- `x0` is an vector/array of initial values at the time `ts(1)`.
- `ts` is a vector of meso-scale times to compute the approximate solution, say in $\mathbb{R}^\ell$ for $\ell \geq 2$ .
- `nMicro`, optional, default 10, is the number of micro-time-steps taken for each meso-scale time-step.
- `patches` struct set by `configPatches`$n$ and provided as either as parameter, or as a global variable.

**Output**

- `xs`, 5/7/9D (depending upon `nD`) array of length $\ell \times \cdots$ of approximate solution vector/array at the specified times. But, if using parallel computing via `spmd`, then `xs` is a *composite* 5/7/9D array, so outside of an `spmd`-block access a single copy of the array via `xs{1}`. Similarly for `errs`.
- `errs`, column vector in $\mathbb{R}^\ell$ of local error estimate for the step from $t_{k-1}$ to $t_k$.

# Bibliography

Bunder, J., Divahar, J., Kevrekidis, I. G., Mattner, T. W. & Roberts, A. (2019), Large-scale simulation of shallow water waves with computation only on small staggered patches, Technical report, https://arxiv.org/abs/1912.07815.

Bunder, J. E., Kevrekidis, I. G. & Roberts, A. J. (2020), Equation-free patch scheme for efficient computational homogenisation via self-adjoint coupling, Technical report, http://arxiv.org/abs/2007.06815.

Bunder, J. E. & Roberts, A. J. (2018), Nonlinear emergent macroscale PDEs, with error bound, for nonlinear microscale systems, Technical report, [https://arxiv.org/abs/1806.10297].

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

Bunder, J., Roberts, A. J. & Kevrekidis, I. G. (2016), 'Accuracy of patch dynamics with mesoscale temporal coupling for efficient massively parallel simulations', *SIAM Journal on Scientific Computing* **38**(4), C335–C371.

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

Dellar, P. J. & Salmon, R. (2005), 'Shallow water equations with a complete coriolis force and topography', *Phys. Fluids* **17**, 106601.

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.

Hereman, W. (2009), Shallow water waves and solitary waves, *in* 'Mathematics of Complexity and Dynamical Systems', Springer, New York, pp. 8112–8125.

Hyman, J. M. (2005), 'Patch dynamics for multiscale problems', *Computing in Science & Engineering* **7**(3), 47–53.
http://scitation.aip.org/content/aip/journal/cise/7/3/10.1109/MCSE.2005.57

Kevrekidis, I. G., Gear, C. W. & Hummer, G. (2004), 'Equation-free: the computer-assisted analysis of complex, multiscale systems', *A. I. Ch. E. Journal* **50**, 1346–1354.

Kevrekidis, I. G., Gear, C. W., Hyman, J. M., Kevrekidis, P. G., Runborg, O. & Theodoropoulos, K. (2003), 'Equation-free, coarse-grained multiscale computation: enabling microscopic simulators to perform system level tasks', *Comm. Math. Sciences* **1**, 715–762.

Kevrekidis, I. G. & Samaey, G. (2009), 'Equation-free multiscale computation: Algorithms and applications', *Annu. Rev. Phys. Chem.* **60**, 321—44.

Leitenmaier, L. & Runborg, O. (2021), Heterogeneous multiscale methods for the landau-lifshitz equation, Technical report, http://arxiv.org/abs/2108.09463.

Liu, P., Samaey, G., Gear, C. W. & Kevrekidis, I. G. (2015), 'On the acceleration of spatially distributed agent-based computations: A patch dynamics scheme', *Applied Numerical Mathematics* **92**, 54–69.
http://www.sciencedirect.com/science/article/pii/S0168927414002086

MacKenzie, T. & Roberts, A. J. (2003), Holistic discretisation of shear dispersion in a two-dimensional channel, *in* K. Burrage & R. B. Sidje, eds, 'Proc. of 10th Computational Techniques and Applications Conference CTAC-2001', Vol. 44, pp. C512–C530.

Maclean, J., Bunder, J. E. & Roberts, A. J. (2020), 'A toolbox of equation-free functions in matlab/octave for efficient system level simulation', *Numerical Algorithms* .

Maclean, J. & Gottwald, G. A. (2015), 'On convergence of higher order schemes for the projective integration method for stiff ordinary differential equations', *Journal of Computational and Applied Mathematics* **288**, 44–69.
http://www.sciencedirect.com/science/article/pii/S0377042715002149

Maier, B. & Verfürth, B. (2021), Numerical upscaling for wave equations with time-dependent multiscale coefficients, Technical report, http://arxiv.org/abs/2107.14069.

Marschler, C., Sieber, J., Berkemer, R., Kawamoto, A. & Starke, J. (2014), 'Implicit methods for equation-free analysis: Convergence results and analysis of emergent waves in microscopic traffic models', *SIAM J. Appl. Dyn. Syst.* **13**(2), 1202–1238.

Oron, A., Davis, S. H. & Bankoff, S. G. (1997), 'Long-scale evolution of thin liquid films', *Rev. Mod. Phys.* **69**, 931–980. http://link.aps.org/abstract/RMP/v69/p931.

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

Roberts, A. J. (2003), 'A holistic finite difference approach models linear dynamics consistently', *Mathematics of Computation* **72**, 247–262. http://www.ams.org/mcom/2003-72-241/S0025-5718-02-01448-5

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

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

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

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

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

Taylor, G. I. (1953), 'Dispersion of soluble matter in solvent flowing slowly through a tube', *Proc. Roy. Soc. Lond. A* **219**, 186–203.

Watt, S. D. & Roberts, A. J. (1995), 'The accurate dynamic modelling of contaminant dispersion in channels', *SIAM J. Appl. Math.* **55**(4), 1016–1038. http://epubs.siam.org/sam-bin/dbq/article/25797.

Wilson, S. K., Hunt, R. & Duffy, B. R. (2000), 'The rate of spreading in spin coating', *J. Fluid Mech.* **413**, 65–88.