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

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

April 5, 2019

[*] 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

[§] Appear here for your contribution.

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. The methodology bypasses the derivation of macroscopic evolution equations by using only short bursts of microscale simulations which are often the best available description of a system (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.

Contents

1 Introduction

Users 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 start, example code of their use (code which is executed if the function is invoked without any arguments).

- To projectively integrate over time a multiscale, slow-fast, system of ODEs you could use `PIRK2()`: adapt the Michaelis–Menten example at the start 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 start of `PIG.m` (Section 2.4.2).
- Consider an evolving system over a large spatial domains 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 adapt the code at the start of `configPatches1.m` for Burgers' PDE (Section 3.2.2);
  - in 2D adapt the code at the start of `configPatches2.m` for non-linear diffusion (Section 3.6.2).
- The above are for systems that have *smooth* spatial structures on the microscale: when the microscale is 'rough' with a known period (so far only in 1D), then adapt the example of `HomogenisationExample.m` (Section 3.5).

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 any $p = 1, 2, \ldots, \infty$, and evolving time t. The details $\vec{u}$ could represent particles, agents, 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. 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); in particle problems the positions would evolve. And suppose your detailed and trustworthy simulation is coded 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 patch. The computational savings may be enormous, especially if combined with projective integration.

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

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

2 Projective integration of deterministic ODEs

Chapter contents

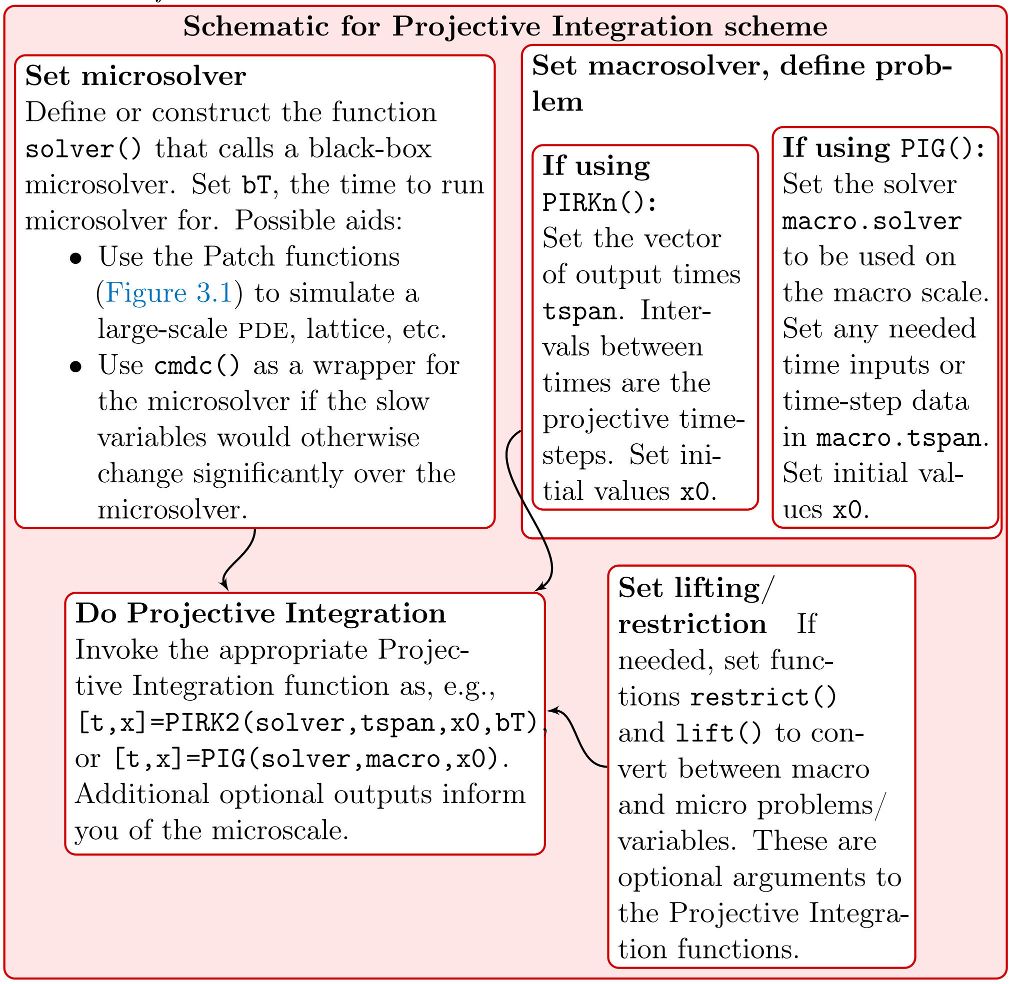
2.1 Introduction

This section provides some good projective integration functions (Gear & Kevrekidis 2003*b*,*c*, Givon et al. 2006, Maclean & Gottwald 2015, Sieber et al. 2018, e.g.). The goal is to enable computationally expensive multi-scale 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 simulation of the complex system, and call such code a microsolver.

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

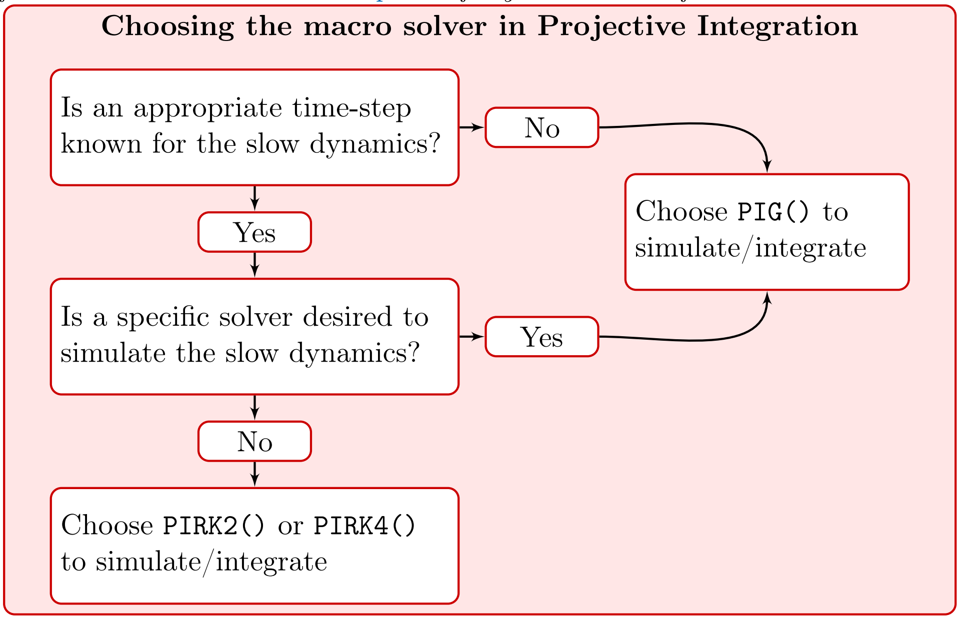


The Projective Integration section of this toolbox consists of several functions. Each function implements over a long-time scale a variant of a standard numerical method to simulate/integrate the emergent dynamics of the complex system. Each function has standardised inputs and outputs.

Main functions

- Projective Integration by second or fourth-order Runge–Kutta, `PIRK2()` and `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 solver, `PIG()`. This function

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.



enables a Projective Integration implementation of any solver with macroscale time-steps. It does not matter whether the solver is a standard Matlab algorithm, or one supplied by the user. As explored in later examples, `PIG()` should only be used in very stiff systems.

- 'Constraint-defined manifold computing', `cdmc()`. This helper function, based on the method introduced in Gear et al. (2005*a*), iteratively applies the microsolver and projects the output backwards in time. The result is to constrain the fast variables close to the slow manifold, without advancing the current time by the duration of an application of the microsolver. This function can be used to reduce errors related to the simulation length of the microsolver in either the `PIRK` or `PIG` functions. In particular, it enables `PIG()` to be used on problems that are not particularly stiff.

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

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

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

Section contents

2.2.1 Introduction

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

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

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

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

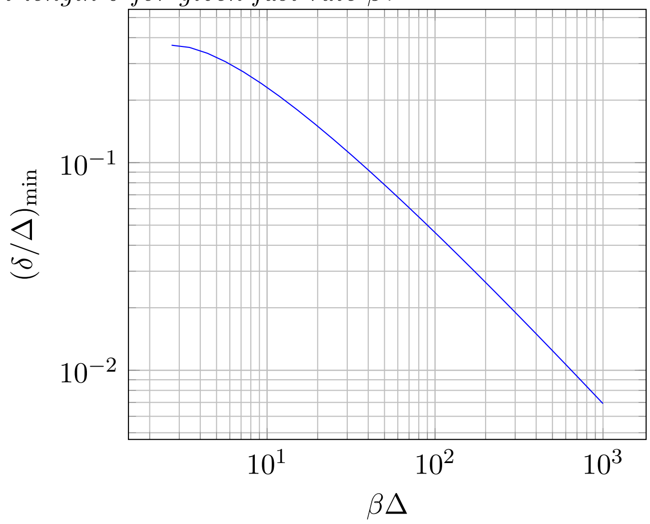
  `[tOut, xOut] = microBurst(tStart, xStart, bT)`

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

- `tSpan` is an ℓ-vector of times at which the user requests output, of which the first element is always the initial time. `PIRK2()` does not use adaptive time-stepping; the macroscale time-steps are (nearly) the steps between elements of `tSpan`.
- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: they are included in the simulation and output, and often represent boundaries in space fields.
- `bT`, optional, either missing, or empty (`[]`), or a scalar: if a given scalar, then it is the length of the micro-burst simulations—the minimum amount of time needed for the microscale simulation to relax to the slow manifold; else if missing or `[]`, then `microBurst()` must itself determine the length of a computed burst.

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

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



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

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 two to four optional outputs of the microscale bursts.

- `tms`, optional, is an L dimensional column vector containing microscale times of burst simulations, each burst separated by `NaN`;
- `xms`, optional, is an $L \times n$ array of the corresponding microscale states—this data is an accurate simulation of the state and may help visualise more 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 in general resemble the true dynamics.

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

2.2.2 If no arguments, then execute an example

```
if nargin==0
```

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

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

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

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

Upon finishing execution of the example, exit this function.

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

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

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

Section contents

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

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

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

2.3.1 Invoke projective integration

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

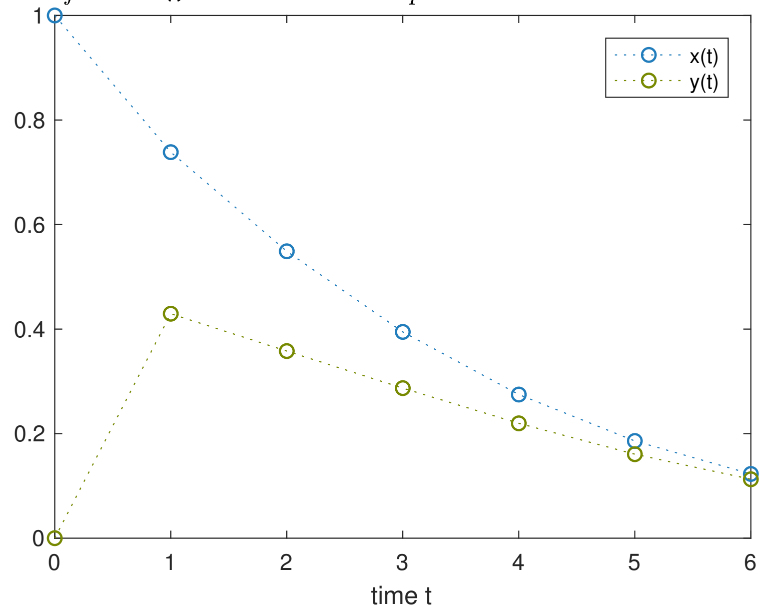
```
clear all, close all
global MMepsilon
MMepsilon = 0.1
```

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

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

Figure 2.4 plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold. Sieber et al. (2018) [§4] used

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



this system as an example of their analysis of the convergence of Projective Integration.

Optional: 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). 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.

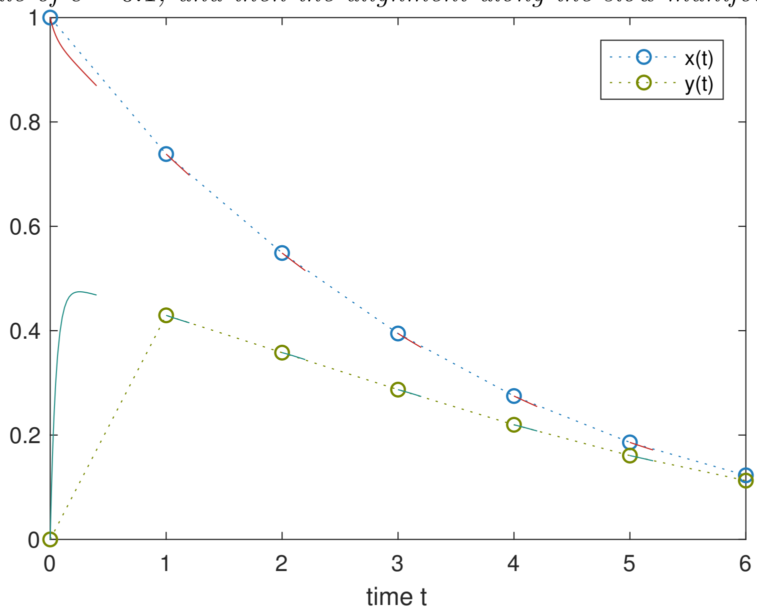
```
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, [1;0], 2*MMepsilon);
figure, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
pause(1)
```

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

Optional: 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*). Such backward macroscale simulations succeed despite the fast variable $y(t)$, when backward in time, being viciously unstable. However, backward integration appears to need longer bursts, here 3ϵ.

```
ts = 0:-1:-5
[xs,tMicro,xMicro] = PIRK2(@MMburst, ts, 0.2*[1;1], 3*MMepsilon);
```

Figure 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, plot(ts,xs,'o:',tMicro,xMicro)
xlabel('time t'), legend('x(t)','y(t)')
```

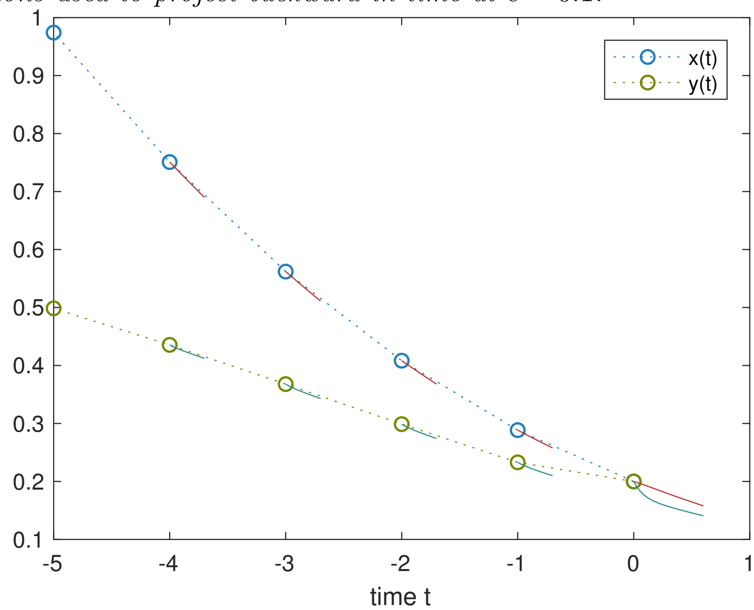
Code a burst of Michaelis–Menten enzyme kinetics Integrate a burst of length `bT` of the ODEs for the Michaelis–Menten enzyme kinetics at parameter ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), we here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

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

Section contents

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



2.4.1 Introduction

This is a Projective Integration scheme when the macroscale integrator is any specified coded scheme. 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, `PIG()` uses 'constraint-defined manifold computing' for the microscale simulations. This algorithm, initiated by Gear et al. (2005*b*), uses a backwards projection so that the simulation time is unchanged after running the microscale simulator. The implementation is `cdmc()`, described in Section 2.5.2.

```
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 input a standard MATLAB/Octave integration function (such as `'ode23'` or `'ode45'`), or code your own integration function using standard arguments. That is, if you code your own, then it must be

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

  where

  – function `F(T,X)` notionally evaluates the time derivatives $d\vec{X}/dt$ at any time;

  - `Tspan` is either the macro-time interval, or the vector of macroscale times at which macroscale values are to be returned; and
  - `X0` are the initial values of $\vec{X}$ at time `Tspan(1)`.

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

- `microBurst()` is a function that produces output from the user-specified code for a burst of microscale simulation. The function must internally specify 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` adaptively selects 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 n-dimensional microscale state $\vec{x}$ and computes the corresponding N-dimensional macroscale state $\vec{X}$;

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

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

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

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

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

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

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

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

- `tms`, optional, is an ℓ-dimensional column vector containing microscale times of burst simulations, 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
```

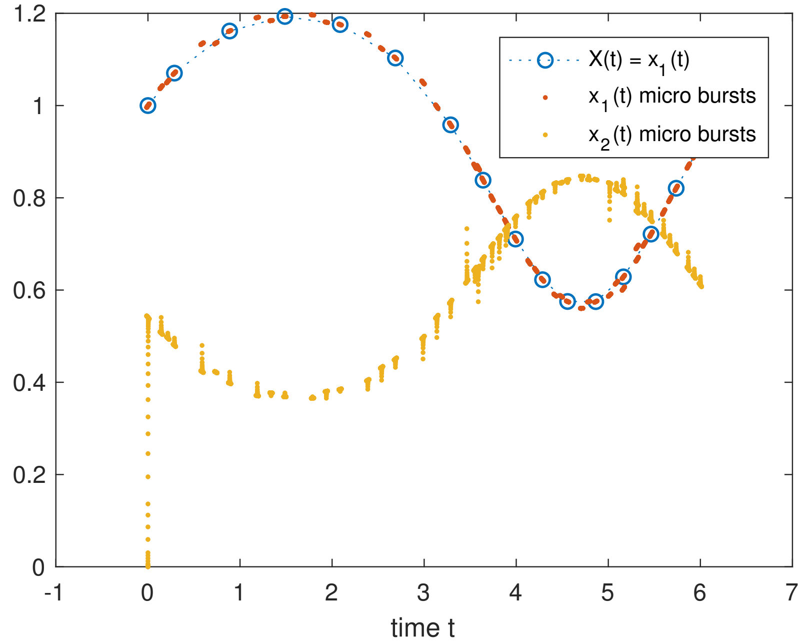
As a basic example, consider a microscale system of the singularly perturbed system of differential equations

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

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.

Figure 2.7: Projective Integration by **PIG** *of the example system* (2.1) *in 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.*




```
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 $\beta \approx 1/\epsilon$ and 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 `ode23()`, 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*).

```
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 $t = -5$. 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 ϵ inherited from above. Code ODEs in function `dMMdt` with variables $x =$ `x(1)` and $y =$ `x(2)`. Starting at time `ti`, and state `xi` (row), we here simply use `ode23` to integrate 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
```

2.5.2 `cdmc()`

`cdmc()` iteratively applies the micro-burst and then projects backwards in time to the initial conditions. 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

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 can be simulated by the microBurst `sol(t,x,T)`, one would define

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

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

3 Patch scheme for given microscale discrete space system

Chapter contents

3.1 Introduction

The patch scheme applies to spatio-temporal systems where the spatial domain is larger than what can be computed in reasonable time in a given complicated microscale code. In the scheme we compute 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); 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 approximation of a PDE. Usually continuous in time.

Quick start See Sections 3.2.2 and 3.6.2 which respectively list example basic code that uses the provided functions to simulate the 1D Burgers' PDE, and a 2D nonlinear 'diffusion' PDE. Then see Figure 3.1.

3.2 `configPatches1()`: configures spatial patches in 1D

Section contents

3.2.1 Introduction

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

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

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

- `fun` is the name of the user function, `fun(t,u,x)`, that computes time derivatives (or time-steps) of quantities on the patches.
- `Xlim` give the macro-space 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 domain.
- `nPatch` is the number of equi-spaced spaced patches.
- `ordCC` is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: currently must be ≥ -1.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; and `ratio` $= 1$ is overlapping patches as in holistic discretisation.
- `nSubP` is the number of equi-spaced microscale lattice points in each patch. Must be odd so that there is a central lattice point.

Figure 3.1: The Patch methods, Chapter 3, accelerate simulation/integration of multiscale systems with interesting spatial (or network) structure/patterns. The patch 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 functions involved and their interrelationships.

Patch scheme for PDEs

Define problem and construct patches
Invoke `configpatches1` (for 1D) or `configpatches2` (for 2D) to define the microscale problem (PDE, domain, boundary conditions, etc) and the desired patch structure (number of patches, patch size, coupling order, etc). These functions initialise the global struct `patches`. The components of `patches` contain all information required to solve the microscale problem within each patch. If necessary, define additional components for struct `patches` (e.g., see `EnsembleAverageExample.m`).

Solve microscale problem within each patch
Call the PDE solver which is to evaluate the microscale problem within each patch. This solver may be a Matlab defined function (such as `ode15s` or `ode45`) or a user defined function (such as Runge–Kutta).
Input of the PDE solver is the function `patchSmooth1` (for 1D) or `patchSmooth2` (for 2D) which interfaces with the PDE solver and the microscale PDE. Other inputs are the time span and initial conditions. Output of the PDE solver is the solution of the patch PDE over the given time span, but only evaluated within the defined patches.

Projective integration scheme (if needed)

Interface to time integrators
The PDE function (`patchSmooth1` or `patchSmooth2`) interfaces with the PDE solve, the microscale PDE and the patch coupling conditions. Input is the PDE field at one time-step and output is the field at the next time-step.

Coupling conditions
Coupling conditions are evaluated in `patchEdge1` (for 1D) or `patchEdge2` (for 2D) with the coupling order defined by global struct component `patches.ordCC`.

Microscale PDE
This PDE is defined by the global struct `patches`, for example component `patches.fun` defines the function (e.g., `BurgersPDE` or `heteroDiff`) and `patches.x` defines the domain of the patches

Process results and plot

- `nEdge`, optional, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbour interactions).

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

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

3.2.2 If no arguments, then execute an example

```
if nargin==0
```

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

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

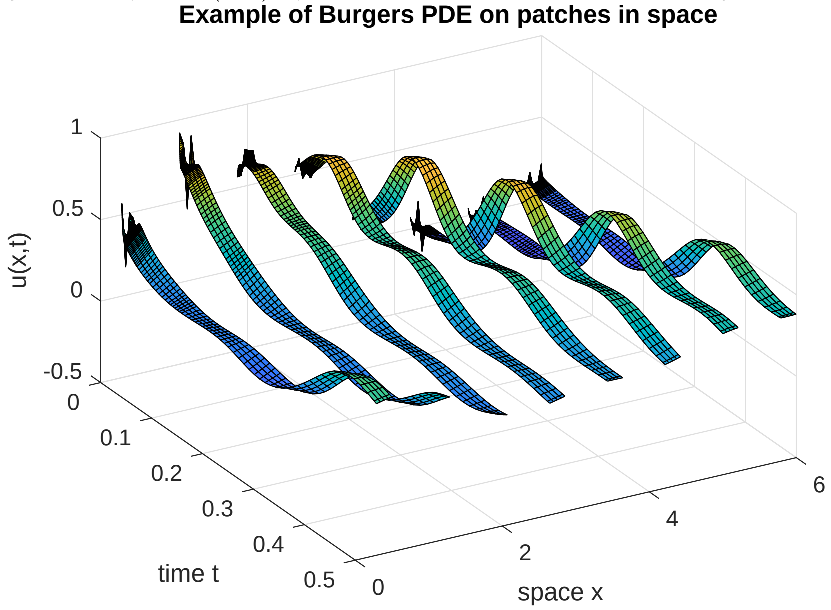
Establish global patch data struct to interface with a function coding Burgers' PDE: to be solved on 2π-periodic domain, with eight patches, spectral interpolation couples the patches, each patch of half-size ratio 0.2, and with seven microscale points forming each patch.

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

Set an initial condition, with some randomness, and simulate in time using a standard stiff integrator and the interface function `patchsmooth1()` (Section 3.3).

```
u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
if ~exist('OCTAVE_VERSION','builtin')
[ts,ucts] = ode15s( @patchSmooth1,[0 0.5],u0(:));
else % octave version
[ts,ucts] = odeOcts(@patchSmooth1,[0 0.5],u0(:));
end
```

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



Plot the simulation using only the microscale values interior to the patches: set x-edges to `nan` to leave the gaps. Figure 3.2 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```
figure(1),clf
patches.x([1 end],:)=nan;
surf(ts,patches.x(:),ucts'), view(60,40)
title('Example of Burgers PDE on patches in space')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
```

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 $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$.

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

3.3 `patchSmooth1()`: interface to time integrators

Section contents

3.3.1 Introduction

To simulate in time with spatial patches we often need to interface a user's time derivative function with time integration routines such as `ode15s` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables to this function using the previously established global struct `patches` (Section 3.2).

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

Input

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

Output

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

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

Section contents

3.4.1 Introduction

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation of either the mid-patch value or the patch-core average. This function is primarily used by `patchSmooth1()` but is also useful for user graphics. A spatially discrete system could be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Assumes that the core averages are in some sense *smooth* so that these averages are sensible macroscale variables. Then patch edge values are determined by macroscale interpolation of the core averages (Bunder et al. 2017). Communicate patch-design variables via the global struct `patches`.

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

Input

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

Output

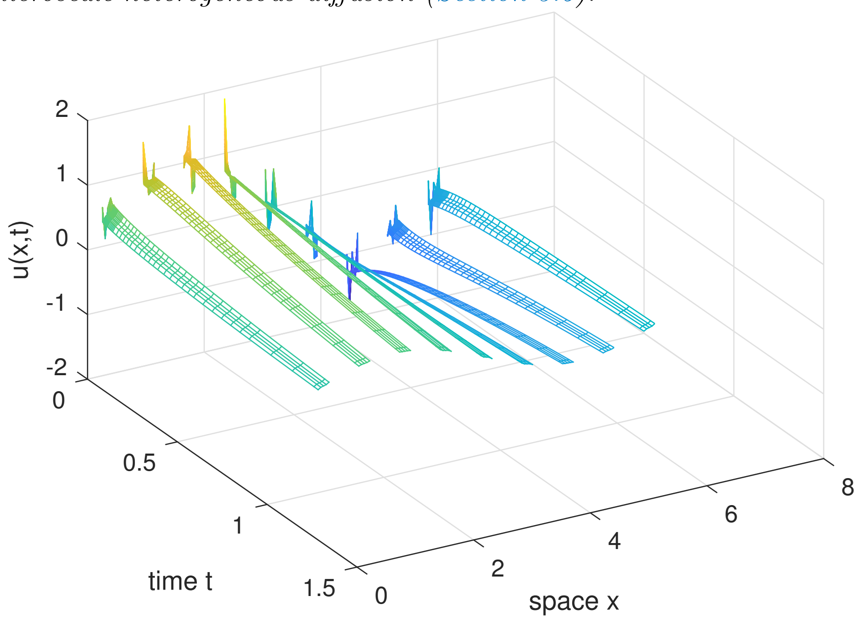
- `u` is `nSubP` $\times$ `nPatch` $\times$ `nVars` 2/3D array of the fields with edge values set by interpolation of patch core averages.

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

Section contents

Figure 3.3 shows an example simulation in time generated by the patch scheme function 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 microscale periodicity.

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



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

1. configPatches1
2. ode15s ↔ patchSmooth1 ↔ heteroDiff
3. process results

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

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

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

3.5.1 Script to simulate via stiff or projective integration

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

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

Establish global data struct `patches` for heterogeneous diffusion solved on 2π-periodic domain, with nine patches, each patch of half-size 0.2. A user can add information to `patches` in order to communicate to the time derivative function. Quadratic (fourth-order) interpolation $\texttt{ordCC} = 4$ provides values for the inter-patch coupling conditions. The odd integer $\texttt{patches.nCore} = 3$ defines the size of the patch core (this must be larger than zero and less than nSubP), where a core of size zero indicates that the value in the centre of the patch gives the macroscale. The introduction of a finite width core requires a redefinition of the half-patch ratio, as described by Bunder et al. (2017). We evaluate the patch coupling by interpolating the core.

```
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);
```

Add to the global data struct `patches` for use by the time derivative function (for example): here include the diffusivity coefficients, repeated to fill up a patch

```
patches.c=repmat(cDiff,(nSubP-1)/mPeriod,1);
```

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

```
u0 = sin(patches.x)+0.4*randn(nSubP,nPatch);
if ~exist('OCTAVE_VERSION','builtin')
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
else % octave version
[ts,ucts] = odeOcts(@patchSmooth1, [0 2/cHomo], u0(:));
end
ucts = reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the simulation in Figure 3.3.

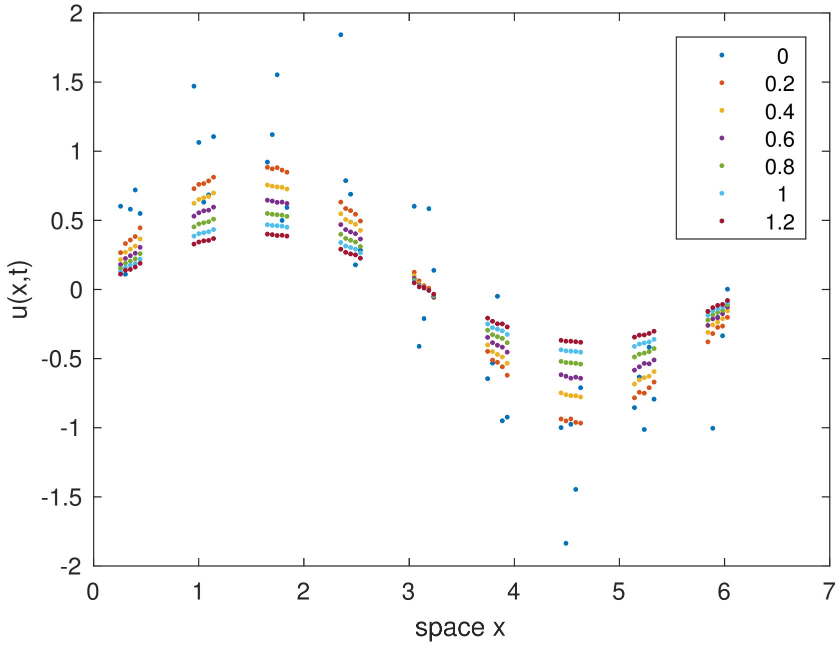
```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),ucts'),  view(60,40)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','homogenisationCtsU')
```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration `PIRK2` (Section 2.2), of bursts of simulation from `heteroBurst` (Section 3.5.3), as illustrated by Figure 3.4.

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

1. configPatches1 (done in first part)

Figure 3.4: 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.



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

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

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

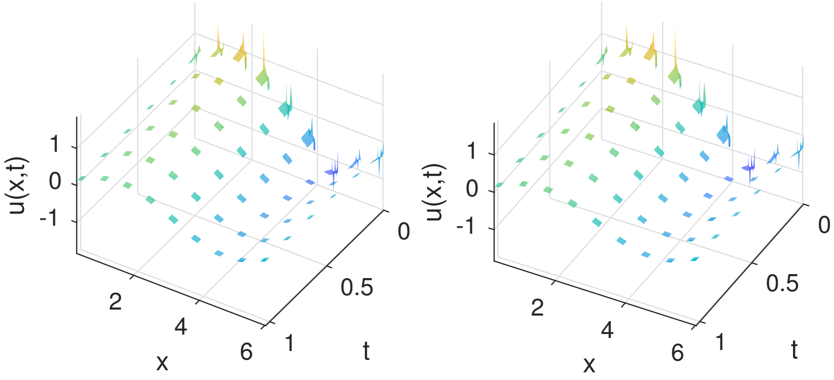
Set the desired macro- and microscale time-steps over the time domain: the macroscale step is in proportion to the effective mean diffusion time on the macroscale; the burst time is proportional to the intra-patch effective diffusion time; and lastly, the microscale time-step is proportional to the diffusion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

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

Plot the macroscale predictions to draw Figure 3.4.

```
figure(2),clf
plot(xs(:),us','.')
ylabel('u(x,t)'), xlabel('space x')
legend(num2str(ts',3))
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','homogenisationU')
```

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



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

```
figure(3),clf
for k = 1:2, subplot(1,2,k)
  surf(tss,xs(:),uss',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','homogenisationMicro')
```

End of the script.

3.5.2 `heteroDiff()`: heterogeneous diffusion

This function codes the lattice heterogeneous diffusion inside the patches. For 2D input arrays `u` and `x` (via edge-value interpolation of `patchSmooth1`, Section 3.3), computes the time derivative (3.1) at each point in the interior of a patch, output in `ut`. The column vector (or possibly array) of diffusion coefficients c_i have previously been stored in struct `patches`.

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

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

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

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
        if ~exist('OCTAVE_VERSION','builtin')
```

```
        [ts,ucts] = ode23( @patchSmooth1,[ti ti+bT],ui(:));
        else % octave version
        [ts,ucts] = rk2Int(@patchSmooth1,[ti ti+bT],ui(:));
        end
end
```

Fin.

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

Section contents

3.6.1 Introduction

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. Section 3.6.2 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 3.6.2 for the example code.

- `fun` is the name of the user function, `fun(t,u,x,y)`, that computes time derivatives (or time-steps) of quantities on the 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 use the same interval in both directions.
- `BCs` somehow will define the macroscale boundary conditions. Currently, `BCs` is ignored and the system is assumed macro-periodic in the domain.
- `nPatch` determines the number of equi-spaced spaced patches: if scalar, then use the same number of patches in both directions, otherwise `nPatch(1:2)` give the number in each direction.
- `ordCC` is the 'order' of interpolation across empty space of the macroscale mid-patch values to the edge of the patches for inter-patch coupling: currently must be in $\{0\}$.
- `ratio` (real) is the ratio of the half-width of a patch to the spacing of the patch mid-points: so `ratio` $= \frac{1}{2}$ means the patches abut; and `ratio` $= 1$ would be overlapping patches as in holistic discretisation: if scalar, then use the same ratio in both directions, otherwise `ratio(1:2)` give the ratio in each direction.

- `nSubP` is the number of equi-spaced microscale lattice points in each patch: if scalar, then use the same number in both directions, otherwise `nSubP(1:2)` gives the number in each direction. Must be odd so that there is a central lattice point.
- `nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch. May be omitted. The default is one (suitable for microscale lattices with only nearest neighbours. interactions).

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

- `.fun` is the name of the user's function `fun(u,t,x,y)` that computes the time derivatives (or steps) on the patchy lattice.
- `.ordCC` is the specified order of inter-patch coupling.
- `.alt` 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`-vector of weights for the inter-patch interpolation onto the right and left edges (respectively) with patch:macroscale ratio as specified.
- `.x` is `nSubP(1)` $\times$ `nPatch(1)` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.
- `.y` is `nSubP(2)` $\times$ `nPatch(2)` array of the regular spatial locations y_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

3.6.2 If no arguments, then execute an example

```
if nargin==0
```

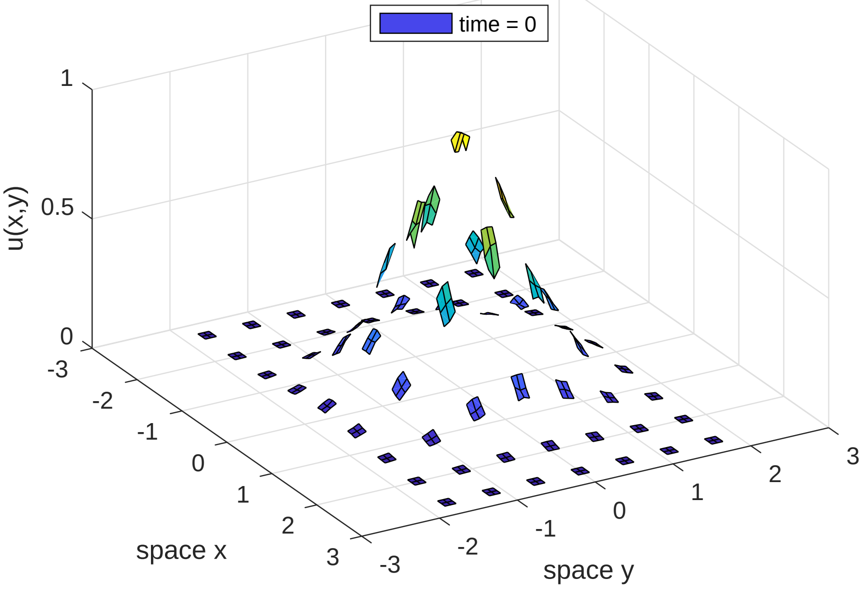
The code here shows one way to get started: a user's script may have the following three steps (arrows indicate function recursion).

1. configPatches2
2. ode15s integrator ↔ patchSmooth2 ↔ user's nonDiffPDE
3. process results

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch. Roberts et al. (2014) established that this scheme is consistent with the PDE (as the patch spacing decreases).

```
nSubP = 5;
configPatches2(@nonDiffPDE,[-3 3 -2 2], nan, [9 7], 0, 0.25, nSubP);
```

Figure 3.6: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 3.7 plots the computed field at time $t = 3$.



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

```
x = reshape(patches.x,nSubP,1,[],1);
y = reshape(patches.y,1,nSubP,1,[]);
u0 = exp(-x.^2-y.^2);
u0 = u0.*(0.9+0.1*rand(size(u0)));
```

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

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

Start by showing the initial conditions of Figure 3.6 while the simulation computes.

```
u = reshape(permute(u0,[1 3 2 4]), [numel(x) numel(y)]);
hsurf = surf(x(:),y(:),u');
axis([-3 3 -3 3 -0.001 1]), view(60,40)
legend('time = 0','Location','north')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
```

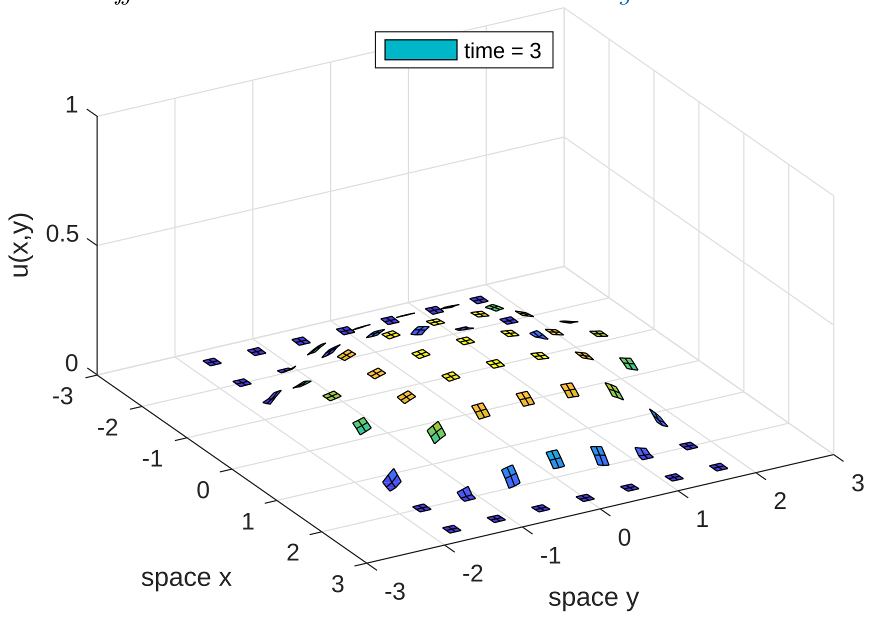
Save the initial condition to file for Figure 3.6.

```
set(gcf,'PaperUnits','centimeters','PaperPosition',[0 0 14 10])
%print('-depsc2','configPatches2ic')
```

Integrate in time using standard functions.

```
disp('Wait while we simulate h_t=(h^3)_xx+(h^3)_yy')
drawnow
```

Figure 3.7: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE *with initial condition in Figure 3.6.*




```
if ~exist('OCTAVE_VERSION','builtin')
        [ts,ucts] = ode15s( @patchSmooth2,[0 3],u0(:));
else % octave version is quite slow
        lsode_options('absolute tolerance',1e-4);
        lsode_options('relative tolerance',1e-4);
        [ts,ucts] = odeOcts(@patchSmooth2,[0 1],u0(:));
end
```

Animate the computed simulation to end with Figure 3.7.

```
for i = 1:length(ts)
  u = patchEdgeInt2(ucts(i,:));
  u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
  set(hsurf,'ZData', u');
  legend(['time = ' num2str(ts(i),2)])
  pause(0.1)
end
%print('-depsc2','configPatches2t3')
```

Upon finishing execution of the example, exit this function.

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

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

```
function ut = nonDiffPDE(t,u,x,y)
  dx = diff(x(1:2));  dy = diff(y(1:2));  % microscale spacing
```

```
  i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior points in patches
  ut = nan(size(u));  % preallocate storage
  ut(i,j,:,:) = diff(u(:,j,:,:).^3,2,1)/dx^2 ...
               +diff(u(i,:,:,:).^3,2,2)/dy^2;
end
```

3.7 `patchSmooth2()`: interface to time integrators

Section contents

3.7.1 Introduction

To simulate in time with spatial patches we often need to interface a users time derivative function with time integration routines such as `ode15s` or `PIRK2`. This function provides an interface. It assumes that the sub-patch structure is *smooth* so that the patch centre-values are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the patch-centre values. Communicate patch-design variables to this function via the previously established global struct `patches`.

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

Input

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

Output

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

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

Section contents

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

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

Input

- `u` is a vector of length `nx` · `ny` · `Nx` · `Ny` · `nVars` where there are `nVars` field values at each of the points in the `nx` × `ny` × `Nx` × `Ny` grid on the `Nx` × `Ny` array of patches.
- `patches` a struct set by `configPatches2()` which includes the following information.
  - `.x` is `nx` × `Nx` array of the spatial locations x_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.y` is similarly `ny` × `Ny` array of the spatial locations y_{ij} of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales.
  - `.ordCC` is order of interpolation, currently only $\{0\}$.
  - `.Cwtsr` and `.Cwtsl`—not yet used

Output

- `u` is `nx` × `ny` × `Nx` × `Ny` × `nVars` array of the fields with edge values set by interpolation.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.