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

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

February 21, 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.). This suite of functions should empower users to start implementing such methods—but so far we have only just started.

Contents

1 Introduction

Subsection contents

Users Place this toolbox's folder in a path searched by MATLAB/Octave. Then read the section that documents the function of interest.

Blackbox scenario Assume that a researcher/practitioner has a detailed and *trustworthy* computational simulation of some problem of interest. The simulation may be written in terms of micro-positional coordinates $\vec{x}_i(t)$ in 'space' at which there are micro-field variable values $\vec{u}_i(t)$ for indices i in some (large) set of integers and for time t. 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. The positional coordinates are $\vec{x}_i \in \mathbb{R}^d$ where for spatial problems integer $d = 1, 2, 3$, but it may be more when solving for a distribution of velocities, or pore sizes, or trader's beliefs, etc. The micro-field variables could be in $\mathbb{R}^p$ for any $p = 1, 2, \ldots, \infty$.

Further, assume that the computational simulation is too expensive over all the desired spatial domain $\mathbb{X} \subset \mathbb{R}^d$. Thus we aim a toolbox to simulate only on macroscale distributed patches.

Contributors The aim of this project is to collectively develop a MATLAB/ Octave toolbox of equation-free algorithms. Initially the algorithms will be simple, and the plan is to subsequently develop more and more capability.

MATLAB appears the obvious choice for a first version since it is widespread, reasonably efficient, supports various parallel modes, and development costs are reasonably low. Further it is built on BLAS and LAPACK so potentially the cache and superscalar CPU are well utilised. Let's develop functions that work for both MATLAB/Octave. **??** outlines some details for contributors.

2 Quick start

Chapter contents

This section may be used in conjunction with the many examples in later sections to help apply the toolbox functions to a particular problem, or to assist in distinguishing between the various functions.

2.1 Cheat sheet: Projective Integration

This section pertains to the Projective Integration (PI) methods of Chapter 3. The PI approach is to greatly accelerate computations of a system exhibiting multiple time scales.

The PI toolbox presents several 'main' functions that could separately be called to perform PI, as well as several optional wrapper functions that may be called. This section helps to distinguish between the top-level PI functions, and helps to tell which of the optional functions may be needed at a glance. Chapter 3 fully details each function.

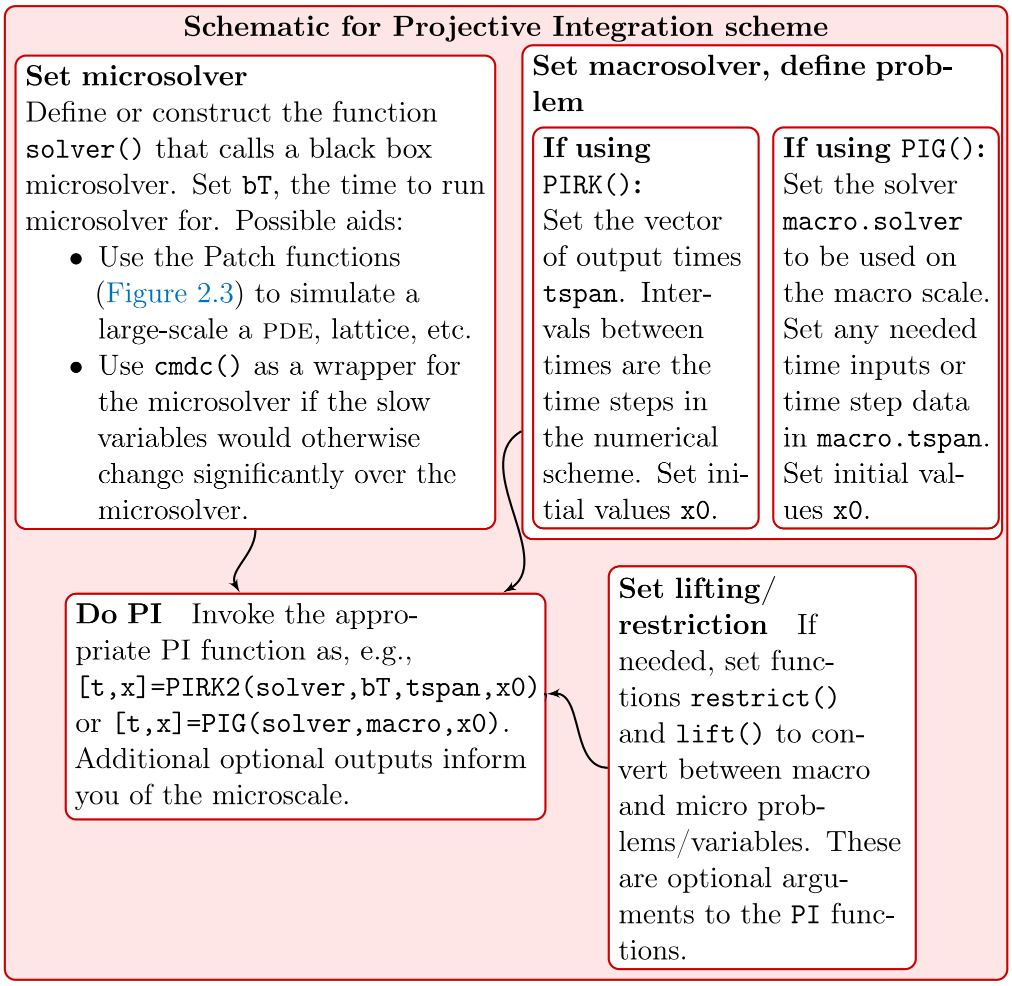
The cheat sheet consists of two flow charts. Figure 2.1 overviews constructing a PI simulation. Figure 2.2 roughly guides which of the top-level PI functions should be used.

2.2 Cheat sheet: constructing patches

This section pertains to the Patch approach, Chapter 4, to solving PDEs, lattice systems, or agent/particle microscale simulators.

The Patch toolbox requires that one configure patches, couple the patches and interface the coupled patches with a time integrator. Figure 2.3 overviews the chief functions involved and their interactions.

Figure 2.1: these figures appear confusing to a newbie???? and we must *not* resize fixed width constructs. Use linewidth for large-scale layout scaling, em for small-widths, and ex for small-heights.



Schematic for Projective Integration scheme

Set microsolver
Define or construct the function `solver()` that calls a black box microsolver. Set `bT`, the time to run microsolver for. Possible aids:

- Use the Patch functions (Figure 2.3) to simulate a large-scale a PDE, lattice, etc.
- Use `cmdc()` as a wrapper for the microsolver if the slow variables would otherwise change significantly over the microsolver.

Set macrosolver, define problem

If using `PIRK()`: Set the vector of output times `tspan`. Intervals between times are the time steps in the numerical scheme. Set initial values `x0`.

If using `PIG()`: Set the solver `macro.solver` to be used on the macro scale. Set any needed time inputs or time step data in `macro.tspan`. Set initial values `x0`.

Do PI Invoke the appropriate PI function as, e.g., `[t,x]=PIRK2(solver,bT,tspan,x0)` or `[t,x]=PIG(solver,macro,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 `PI` functions.

Figure 2.2

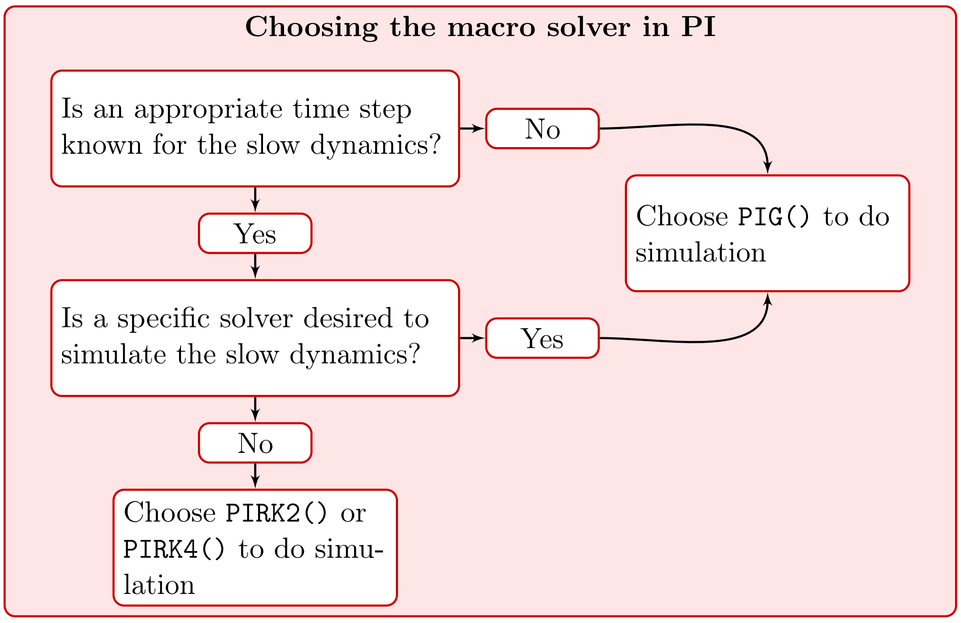
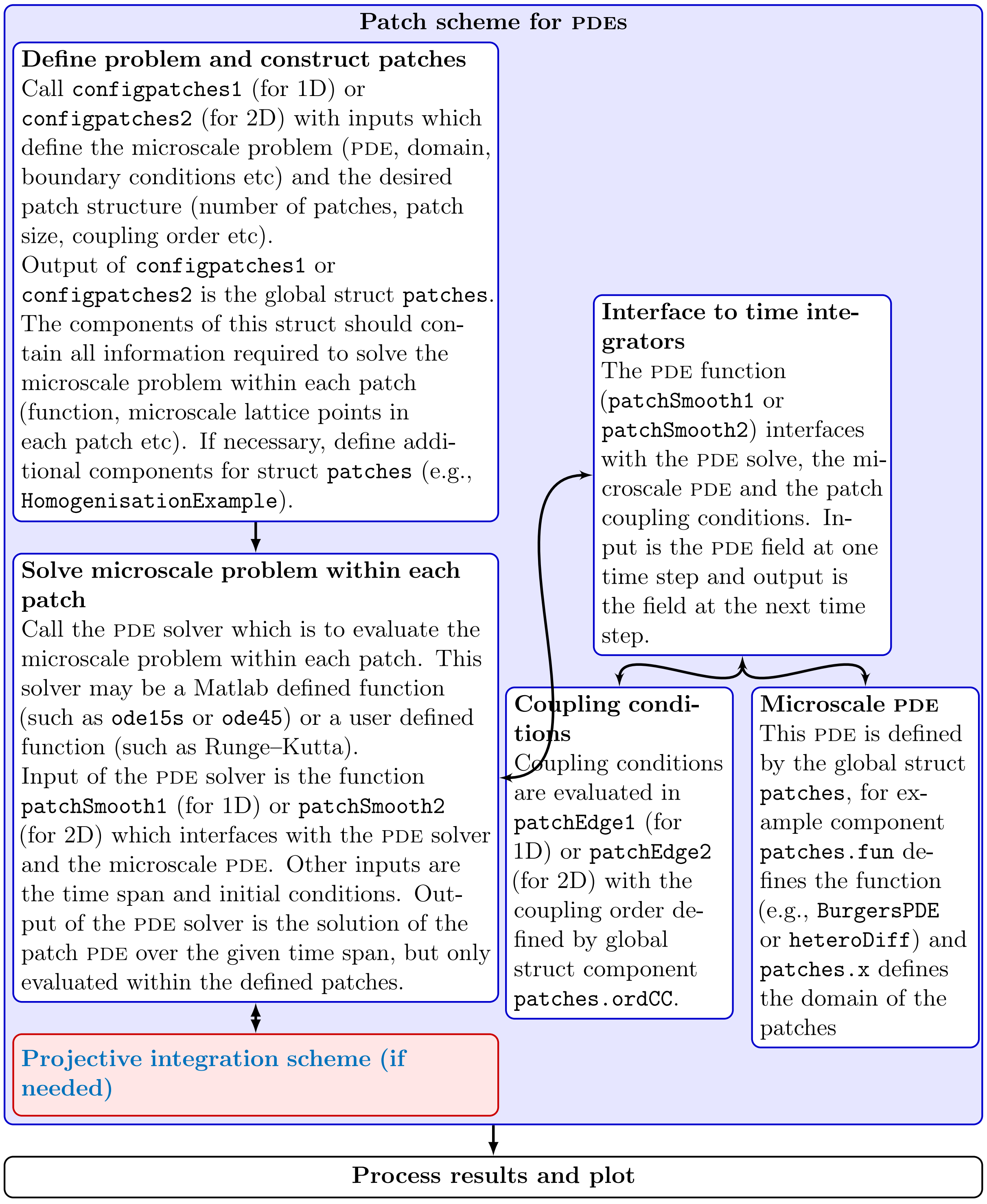

Choosing the macro solver in PI
Is an appropriate time step known for the slow dynamics?
No
Choose PIG() to do simulation
Yes
Is a specific solver desired to simulate the slow dynamics?
Yes
No
Choose PIRK2() or PIRK4() to do simulation

Figure 2.3



Patch scheme for PDEs

Define problem and construct patches
Call `configpatches1` (for 1D) or `configpatches2` (for 2D) with inputs which define the microscale problem (PDE, domain, boundary conditions etc) and the desired patch structure (number of patches, patch size, coupling order etc).
Output of `configpatches1` or `configpatches2` is the global struct `patches`. The components of this struct should contain all information required to solve the microscale problem within each patch (function, microscale lattice points in each patch etc). If necessary, define additional components for struct `patches` (e.g., `HomogenisationExample`).

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

3 Projective integration of deterministic ODEs

Subsection contents

This section provides some good projective integration functions (Gear & Kevrekidis 2003*a*,*b*, Givon et al. 2006, **?**, e.g.). The goal is to enable computationally expensive dynamic simulations to be run over long time scales.

Perhaps start by looking at Section 3.2 which codes the introductory example of a long time simulation of the Michaelis–Menton multiscale system of differential equations.

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.

The Projective Integration section of this toolbox consists of several functions. Each function implements over the long-time scale a variant of a standard numerical method to simulate 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 enables a Projective Integration implementation of any solver with macroscale time steps. It does not matter whether the solver is a standard Matlab or Octave 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 **?**, 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()`.

3.1 `PIRK2()`: projective integration of second order accuracy

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, bT, tSpan, x0)
```

Input If there are no input arguments, then this function applies itself to the Michaelis–Menton example: see the code in Section 3.1.1 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`, the total time to simulate in the burst.
  - Outputs: `tOut`, the column vector of solution times; and `xOut`, an array in which each *row* contains the system state at corresponding times.

- `bT`, a scalar, the minimum amount of time needed for simulation of the microBurst to relax the fast variables to the slow manifold.
- `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.

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

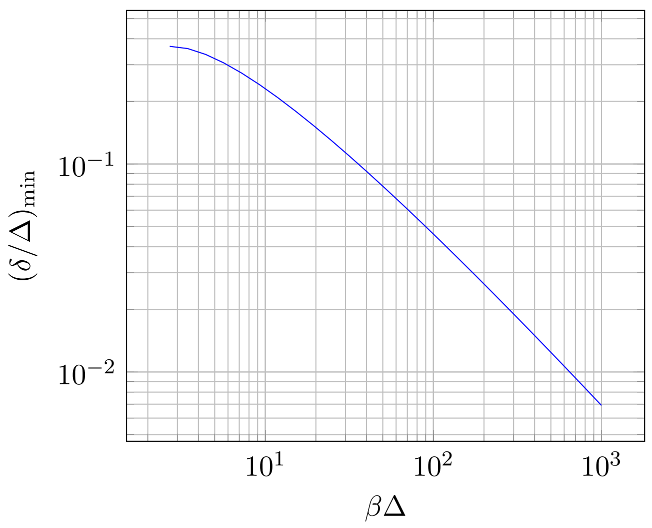
1. a macroscale time step, $\Delta =$ `diff(tSpan)`, such that $\alpha\Delta \approx \sqrt{6\varepsilon}$, and
2. a microscale burst length, $\delta =$ `bT` $\gtrsim \frac{1}{\beta}\log(\beta\Delta)$ (see Figure 3.1).

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,bT,tSpan,x0)`.

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

Figure 3.1: 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.



- `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.

3.1.1 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 $0 \le t \le 6$ 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 $\Delta = 1$.

```
epsilon = 0.05
ts = 0:6
bT = epsilon*log((ts(2)-ts(1))/epsilon)
[x,tms,xms] = PIRK2(@MMburst, bT, ts, [1;0]);
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
```

Example function code for a burst of ODEs 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)
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
            1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
end
```

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

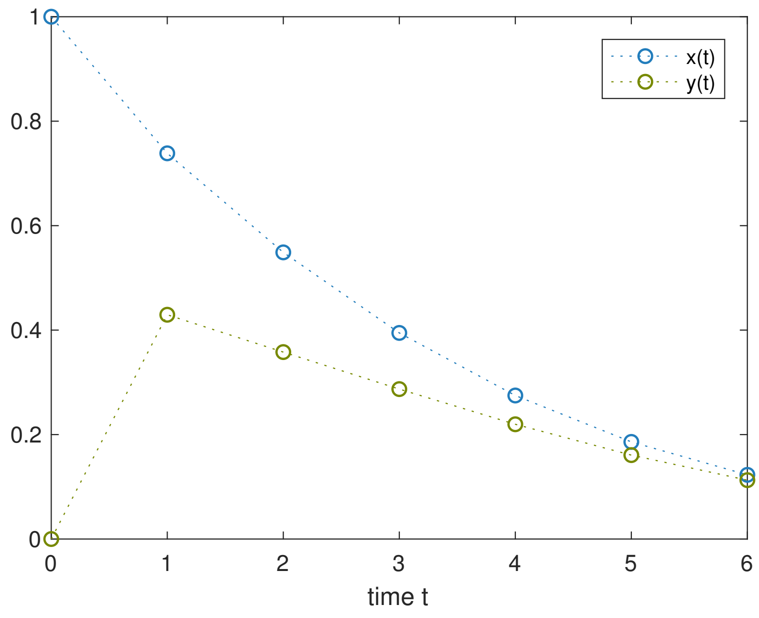
Section contents

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 3.3, 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.

Figure 3.2: Michaelis–Menten enzyme kinetics simulated with the projective integration of `PIRK2()`: macroscale samples.



3.2.1 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 epsilon
epsilon = 0.1
```

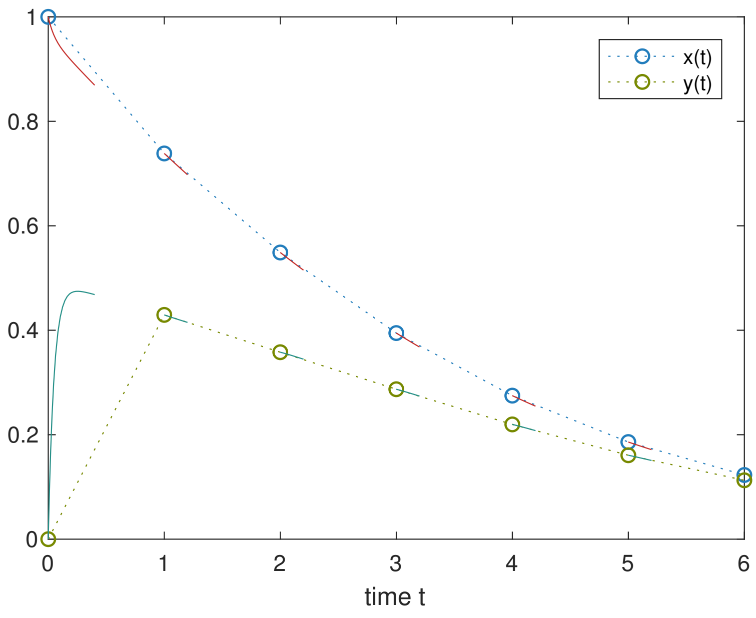
First, Section 3.2.2 encodes the computation of bursts of the Michaelis–Menten system in a function `MMburst()`. Second, here set macroscale times of computation and interest into vector `ts`. Then, invoke Projective Integration with `PIRK2()` applied to the burst function, say using bursts of simulations 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, 2*epsilon, ts, [1;0])
plot(ts,xs,'o:')
xlabel('time t'), legend('x(t)','y(t)')
pause(1)
```

Figure 3.2 plots the macroscale results showing the long time decay of the Michaelis–Menten system on the slow manifold. **?** [§4] used 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 3.2). To see the initial transient attraction to

Figure 3.3: 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.



the slow manifold we plot some microscale data in Figure 3.3. Two further output variables provide this microscale burst information.

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

Figure 3.3 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.

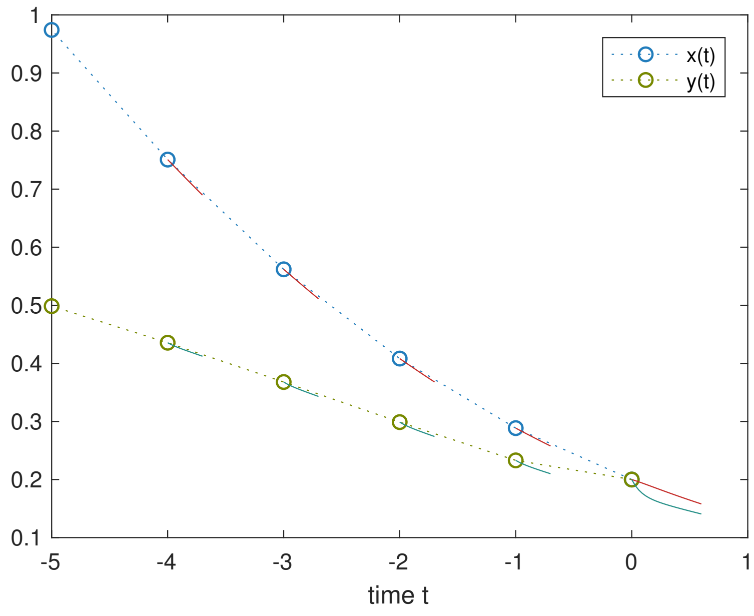
Optional: simulate backwards in time Figure 3.4 shows that projective integration even simulates backwards in time along the slow manifold using short forward bursts. Such backwards macroscale simulations succeed despite the fast variable $y(t)$, when backwards in time, being viciously unstable. However, backwards integration appears to need longer bursts, here 3ϵ.

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

3.2.2 Code a burst of Michaelis–Menten enzyme kinetics

Say use `ode23()` to integrate a burst of the differential equations for the Michaelis–Menten enzyme kinetics. Code differential equations in func-

Figure 3.4: Michaelis–Menten enzyme kinetics simulated backwards with the projective integration of `PIRK2()`: the microscale bursts show the short forward simulations used to project backwards in time at $\epsilon = 0.1$.



tion `dMMdt` with variables x = `x(1)` and y = `x(2)`. For the given start time `ti`, and start state `xi`, `ode23()` integrates the differential equations for a burst time of `bT`, and return the simulation data.

```
function [ts, xs] = MMburst(ti, xi, bT)
    global epsilon
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
          1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
     [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
end
```

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

This is an approximate Projective Integration scheme when the macroscale integrator is any 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 simulation.

Unlike the `PIRKn` functions, `PIG()` does not estimate the slow vector field at the times expected by any user-specified scheme, but instead provides an estimate of the slow vector field at a slightly different time, after an application of the micro-burst simulator. Consequently `PIG()` will incur an additional global error term proportional to the burst length of the microscale simulator. For that reason, `PIG()` should be used with

- either very stiff problems, in which the burst length of the micro-burst can be short,
- or the 'constraint defined manifold' based micro-burst provided by

`cdmc()`, that attempts to project the variables onto the slow manifold without affecting the time.

```
function [t,x,tms,xms,svf] = PIG(macroInt,microBurst,tSpan,x0,lift,restrict)
```

The inputs and outputs are a little different to the two `PIRKn` functions.

Inputs:

- `microBurst()` is a function that produces output from the user-specified code for a burst of microscale simulation. The function must know 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 vector state at the start of a burst.

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

- `macroInt()`, the numerical method that the user wants to apply on a slow-time macroscale. Either use a standard Matlab/Octave integration function (such as `ode23` or `ode45`), or code this solver as a standard Matlab/Octave integration function. That is, if you code you 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 times at which a macroscale value is 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 burst.

- `tSpan`, a vector of times at which the user requests output, of which the first element is always the initial time. If `macroInt` can adaptively select time steps (e.g., `ode45`), then `tSpan` can consist of an initial and final time only.

- `x0`, the vector of initial values at the initial time `tSpan(1)`.

Output If there are no output arguments specified, then a plot is drawn of the computed solution `x` versus `t`. Most often you would only 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 $\ell \times n$ array of the computed solution: the ith *row* of `x`, `x(i,:)`, is to be the vector $\vec x(t)$ at time $t =$ `t(i)`.

  However, microscale details of the underlying Projective Integration computations may be helpful, and so `PIG()` some 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.
- 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 microsolver data to form a macrostep.
  - svf.dx is a $2\ell \times n$ array containing the estimated slow vector field.

3.3.1 If no arguments, then execute an example

```
if nargin==0
```

As a basic example, consider a singularly perturbed system of differential equations for $\vec{x}(t) = (x_1(t), x_2(t))$:

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

With initial conditions $\vec{x}(0) = (1, 0)$, the following code computes and plots a solution of the system over time $0 \le t \le 6$ for parameter $\epsilon = 10^{-3}$.

First we code the right-hand side function of the microscale system 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(). Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $2\epsilon\log(1/\epsilon)$ as here we do not know the macroscale time step invoked by marcoInt(), so blithely use $\Delta = 1$, and then double the usual formula for safety.

```
bT = 2*epsilon*log(1/epsilon)
microBurst = @(tb0,xb0) ode45(dxdt,[tb0 tb0+bT],xb0);
```

Third, invoke PIG to use ode23(), say, on the macroscale slow evolution. Integrate the micro-bursts over $0 \le t \le 6$ from initial condition $\vec{x} = (1, 0)$. (You could set tSpan=[0 -6] to integrate backwards in time with forward bursts.)

```
tSpan = [0 6];
lift = @(x) [x; 0.5];
restrict = @(x) x(1);
[ts,xs,tms,xms] = PIG('ode23',microBurst,tSpan,1, lift, restrict);
```

Plot output of this projective integration.

```
figure, plot(ts,xs,'o:',tms,xms)
title('Projective integration of singular perturbed ODE')
xlabel('time t'), legend('x_1(t)','x_2(t)')
```

Upon finishing execution of the example, exit this function.

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

3.4 `PIRK4()`: projective integration of fourth order accuracy

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

```
function [x, tms, xms, rm, svf] = PIRK4(solver, bT, tSpan, x0)
```

See Section 3.1 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 $-5 \le t \le 0$ for parameter $\epsilon = 0.1$. Since the rate of decay is $\beta \approx 1/\epsilon$ we choose a burst length $\epsilon \log(|\Delta|/\epsilon)$ as here the macroscale time step $\Delta = -1$.

```
epsilon = 0.1
ts = 0:-1:-5
bT = epsilon*log(abs(ts(2)-ts(1))/epsilon)
[x,tms,xms,rm,svf] = PIRK4(@MMburst, bT, ts, 0.2*[1;1]);
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
```

Example function code for a burst of ODEs 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)
    dMMdt = @(t,x) [ -x(1)+(x(1)+0.5)*x(2)
           1/epsilon*( x(1)-(x(1)+1)*x(2) ) ];
    [ts, xs] = ode23(dMMdt, [ti ti+bT], xi);
end
```

3.4.1 `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. `tout(end)` will equal `t`.
- `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.

4 Patch scheme for given microscale discrete space system

Subsection contents

The patch scheme applies to spatio-temporal systems where the spatial domain is larger than what can be computed in reasonable time. Then one may simulate 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 spatial structure is to be on a lattice such as obtained from finite difference approximation of a PDE. Usually continuous in time.

Quick start For an example, see **????** for basic code that uses the provided functions to simulate Burgers' PDE and a nonlinear 'diffusion' PDE.

4.1 configPatches1(): configures spatial patches in 1D

Subsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`. **??** 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 **??** 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 $geq - 1$.
- `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.
- `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 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.

4.2 `patchSmooth1()`: interface to time integrators

Subsection contents

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 using the previously established global struct `patches`.

```
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, but 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.

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

Subsection contents

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation patch core averging. This function is primarily used by `patchSmooth1` but is also useful for user graphics. Consequently a spatially discrete system could be integrated in time via the patch or gap-tooth scheme

(Roberts & Kevrekidis 2007). Assumes that the core averaged structure is *smooth* so that these averages are sensible macroscale variables, and patch edge values are determined by macroscale interpolation of the core averaged values (?). Communicate patch-design variables via the global struct `patches`.

```
function u=patchEdgeInt1(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.
- `patches` a struct set by `configPatches1()` which includes the following.
  - `.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.
  - `.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` × `nPatch` × `nVars` 2/3D array of the fields with edge values set by interpolation of patch core averages.

4.4 `BurgersExample`: simulate Burgers' PDE on patches

Section contents

?? shows an example simulation in time generated by the patch scheme function applied to Burgers' PDE. This code similarly applies the Equation-Free functions to a microscale space-time map (Figure 4.1), a map that happens to be derived as a microscale space-time discretisation of Burgers' PDE. Then this example applies projective integration to simulate further in time.

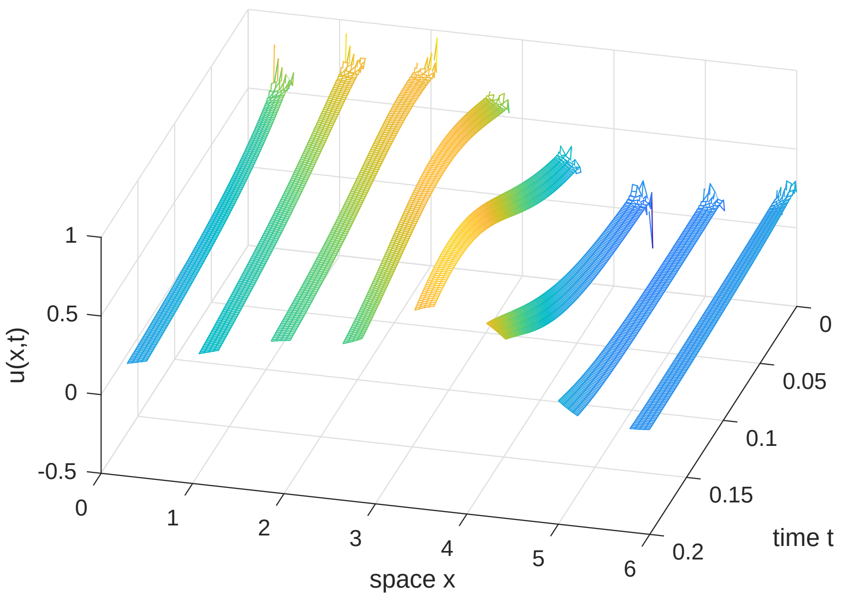
The first part of the script implements the following gap-tooth scheme (arrows indicate function recursion).

1. configPatches1
2. burgerBurst ↔ patchSmooth1 ↔ burgersMap
3. process results

4.4.1 Script code to simulate a microscale space-time map

Establish global data struct for the Burgers' map (Section 4.4.2) solved on 2π-periodic domain, with eight patches, each patch of half-size ratio 0.2, with seven points within each patch, and say fourth order interpolation provides edge-values that couple the patches.

Figure 4.1: a short time simulation of the Burgers' map (Section 4.4.2) on patches in space. It requires many very small time steps only just visible in this mesh.



```
clear all
global patches
nPatch = 8
ratio = 0.2
nSubP = 7
interpOrd = 4
Len = 2*pi
configPatches1(@burgersMap,[0 Len],nan,nPatch,interpOrd,ratio,nSubP);
```

Set an initial condition, and simulate a burst of the microscale space-time map over a time 0.2 using the function `burgerBurst()` (Section 4.4.3).

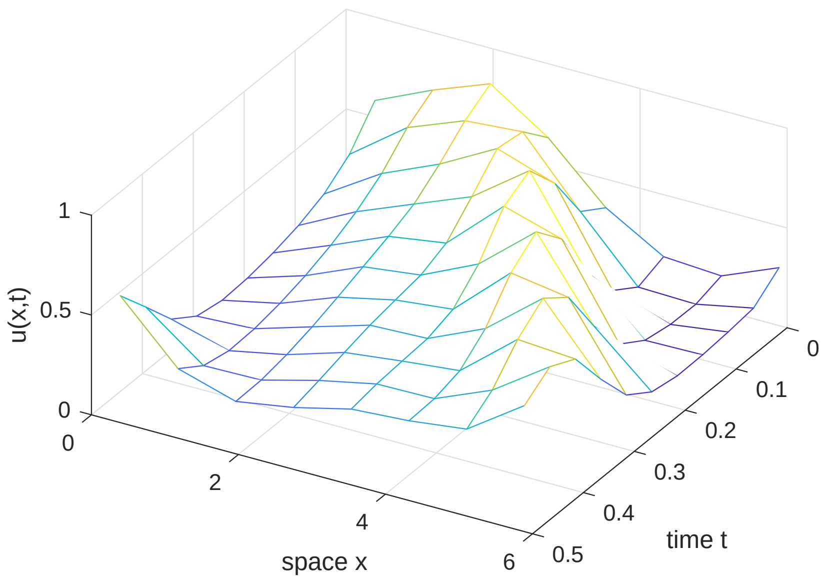
```
u0 = 0.4*(1+sin(patches.x))+0.1*randn(size(patches.x));
[ts,us] = burgerBurst(0,u0,0.2);
```

Plot the simulation. Use only the microscale values interior to the patches via `nan` in the x-edges to leave gaps.

```
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),us')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(105,45)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersMapU')
```

Use projective integration Around the microscale burst `burgerBurst()`, wrap the projective integration function `PIRK2()` of Section 3.1. Figure 4.2 shows the macroscale prediction of the patch centre values on macroscale

Figure 4.2: macroscale space-time field $u(x,t)$ in a basic projective integration of the patch scheme applied to the microscale Burgers' map.



time-steps.

This second part of the script implements the following design.

1. configPatches1 (done in first part)
2. PIRK2 ↔ burgerBurst ↔ patchSmooth1 ↔ burgersMap
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;
```

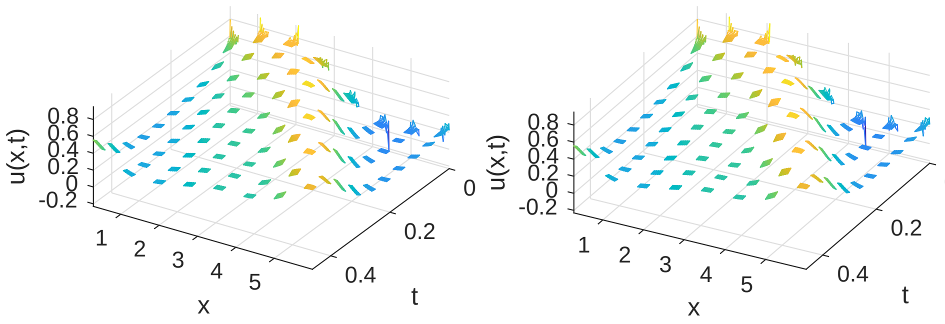
Set the desired macroscale time-steps, and microscale burst length over the time domain. Then projectively integrate in time using `PIRK2()` which is (roughly) second-order accurate in the macroscale time-step.

```
ts = linspace(0,0.5,11);
bT = 3*(ratio*Len/nPatch/(nSubP/2-1))^2
addpath('../ProjInt')
[us,tss,uss] = PIRK2(@burgerBurst,bT,ts,u0(:));
```

Plot the macroscale predictions of the mid-patch values to give the macroscale mesh of Figure 4.2.

```
figure(2),clf
mid = (nSubP+1)/2;
mesh(ts,xs(mid,:),us(:,mid:nSubP:end)')
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
view(120,50)
set(gcf,'paperposition',[0 0 14 10])
print('-depsc2','ps1BurgersU')
```

Figure 4.3: the field $u(x, t)$ during each of the microscale bursts used in the projective integration. View this stereo pair cross-eyed.



Then plot the microscale mesh of the microscale bursts shown in Figure 4.3 (a stereo pair). The details of the fine microscale mesh are almost invisible.

```
figure(3),clf
for k = 1:2, subplot(2,2,k)
  mesh(tss,xs(:),uss')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(126-4*k,50)
end
set(gcf,'paperposition',[0 0 17 12])
print('-depsc2','ps1BurgersMicro')
```

4.4.2 `burgersMap()`: discretise the PDE microscale

This function codes the microscale Euler integration map of the lattice differential equations inside the patches. Only the patch-interior values mapped (`patchSmooth1` overrides the edge-values anyway).

```
function u = burgersMap(t,u,x)
  dx = diff(x(2:3));   dt = dx^2/2;
  i = 2:size(u,1)-1;
  u(i,:) = u(i,:) +dt*( diff(u,2)/dx^2 ...
     -20*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx) );
end
```

4.4.3 `burgerBurst()`: code a burst of the patch map

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

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

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

Apply the microscale map over all time-steps in the burst, using `patchSmooth1` (Section 4.2) as the interface that provides the interpolated edge-values of each patch. Store the results in rows to be consistent with ODE and projective integrators.

```
  us = nan(ndt+1,numel(ui));
  us(1,:) = reshape(ui,1,[]);
  for j = 1:ndt
    ui = patchSmooth1(ts(j),ui);
    us(j+1,:) = reshape(ui,1,[]);
  end
```

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

```
  ts(ndt+1) = ti+bT;
  us(ndt+1,:) = us(ndt,:) ...
    + diff(ts(ndt:ndt+1))/dt*diff(us(ndt:ndt+1,:));
end
```

Fin.

4.5 `HomogenisationExample`: simulate heterogeneous diffusion in 1D on patches

Section contents

Figures 4.4 and 4.5 show example simulations in time generated by the patch scheme function applied to heterogeneous diffusion. That such simulations of heterogeneous diffusion makes valid predictions was established by **?** who proved that the scheme is accurate when the number of points in a patch minus the number of points in the core is an even multiple of the microscale periodicity. We present two different methods of obtaining a macroscale solution. One method uses the given heterogeneous diffusion, which produces a solution which has microscale roughness (Figure 4.4). The other method constructs an ensemble of heterogeneous diffusion and produces an ensemble average solution which has a smooth microscale (Figure 4.5).

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

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

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

$$\dot{u}_i = [c_{i-1/2}(u_{i-1} - u_i) + c_{i+1/2}(u_{i+1} - u_i)]/dx^2. \tag{4.1}$$

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

Figure 4.4: the diffusing field $u(x, t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with no ensemble average. The heterogeneous diffusion results in a smilarly heterogeneous field solution.

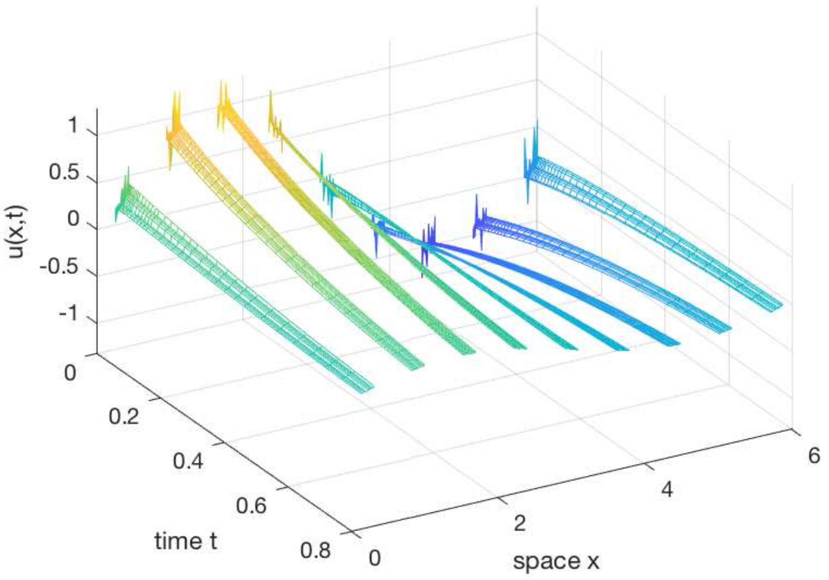
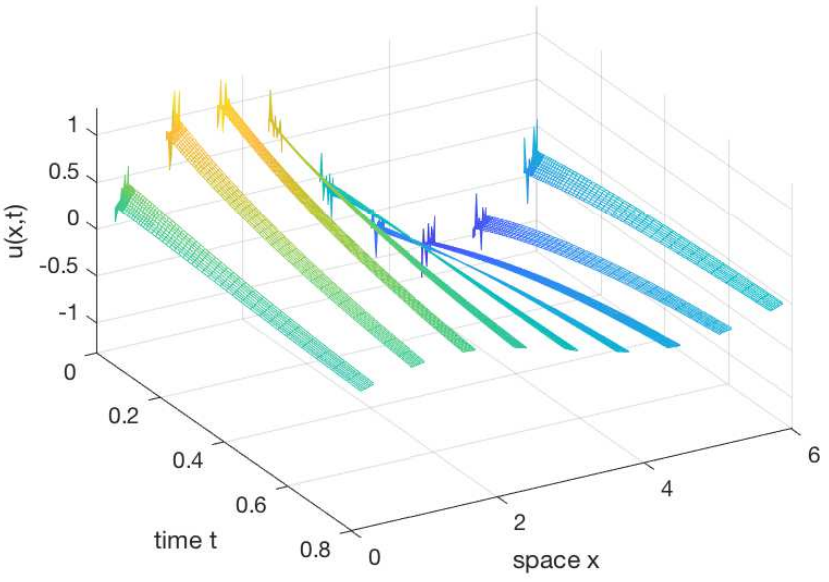


Figure 4.5: the diffusing field $u(x, t)$ in the patch (gap-tooth) scheme applied to microscale heterogeneous diffusion with an ensemble average. The ensemble average smooths out the heterogeneous diffusion.



4.5.1 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 = 4
rng('default'); rng(1);
cDiff = exp(4*rand(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 `ordCC` $= 4$ provides values for the inter-patch coupling conditions. The odd integer `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 **?**. The Boolean `patches.Ens` indicates whether or not we apply ensemble averaging of diffusivity configurations. We evaluate the patch coupling by interpolating the core.

```
global patches
nPatch = 9
ratio = 0.2
nSubP = 11
Len = 2*pi;
ordCC=4;
patches.nCore=3;
patches.ratio = ratio*(nSubP - patches.nCore)/(nSubP - 1);
patches.EnsAve=0;
configPatches1(@heteroDiff,[0 Len],nan,nPatch, ...
 ordCC,patches.ratio,nSubP);
```

A (`nSubP-1`) $\times$ `nPatch` matrix defines the diffusivity coefficients within each patch. In the case of ensemble averaging, `nVars` becomes the size of the ensemble (for the case of no ensemble averaging `nVars` is the number of different field variables, which in this example is `nVars` $= 1$) and we use the ensemble described by **?** which includes all reflected and translated configurations of `patches.cDiff`. With ensemble averaging we must increase the size of the diffusivity matrix to (`nSubP-1`) $\times$ `nPatch` $\times$ `nVars`.

```
patches.cDiff = cDiff((mod(round(patches.x(1:(end-1),:) ...
  /(patches.x(2)-patches.x(1))-0.5),mPeriod)+1));
if patches.EnsAve
  if mPeriod>2
    nVars=2*mPeriod;
  else
    nVars=mPeriod;
  end
  patches.cDiff=repmat(patches.cDiff,[1,1,nVars]);
  for sx=2:mPeriod
    patches.cDiff(:,:,sx)=circshift( ...
      patches.cDiff(:,:,sx-1),[sx-1,0]);
   end;
   if nVars>2
     patches.cDiff(:,:,(mPeriod+1):end)=flipud( ...
       patches.cDiff(:,:,1:mPeriod));
   end;
end
```

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 4.2) to the microscale differential equations.

```
u0 = sin(patches.x)+0.2*randn(nSubP,nPatch);
%u0 = exp(-2*(patches.x-Len/2).^2).*(1+0.1*rand(nSubP,nPatch));
if patches.EnsAve
  u0 = repmat(u0,[1,1,nVars]);
end
[ts,ucts] = ode15s(@patchSmooth1, [0 2/cHomo], u0(:));
ucts=reshape(ucts,length(ts),length(patches.x(:)),[]);
```

Plot the simulation in Figure 4.4 (with no ensemble average) or Figure 4.5 (with an ensemble average). If we have calculated an ensemble of field solutions, then we must first take the ensemble average.

```
if patches.EnsAve % calculate the ensemble average
  uctsAve=mean(ucts,3);
else
  uctsAve=ucts;
end
figure(1),clf
xs = patches.x;  xs([1 end],:) = nan;
mesh(ts,xs(:),uctsAve'),  view(60,40)
xlabel('time t'), ylabel('space x'), zlabel('u(x,t)')
set(gcf,'PaperUnits','centimeters');
set(gcf,'PaperPosition',[0 0 14 10]);
if patches.EnsAve
  print('-depsc2','ps1HomogenisationCtsUEnsAve')
else
  print('-depsc2','ps1HomogenisationCtsU')
end
```

Use projective integration in time Now take `patchSmooth1`, the interface to the time derivatives, and wrap around it the projective integration

Figure 4.6: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with no ensemble average: different colours correspond to the times in the legend. This field solution displays some fine scale heterogeneity due to the heterogeneous diffusion.

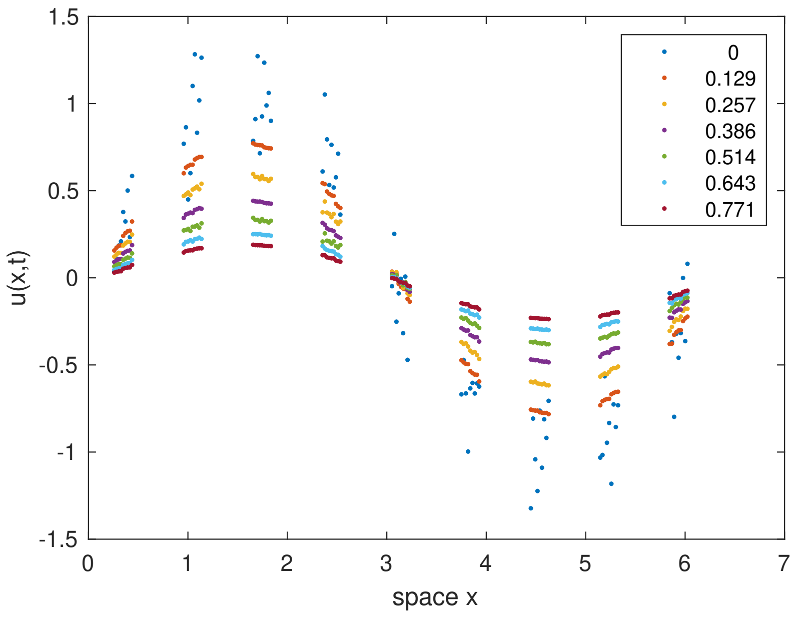
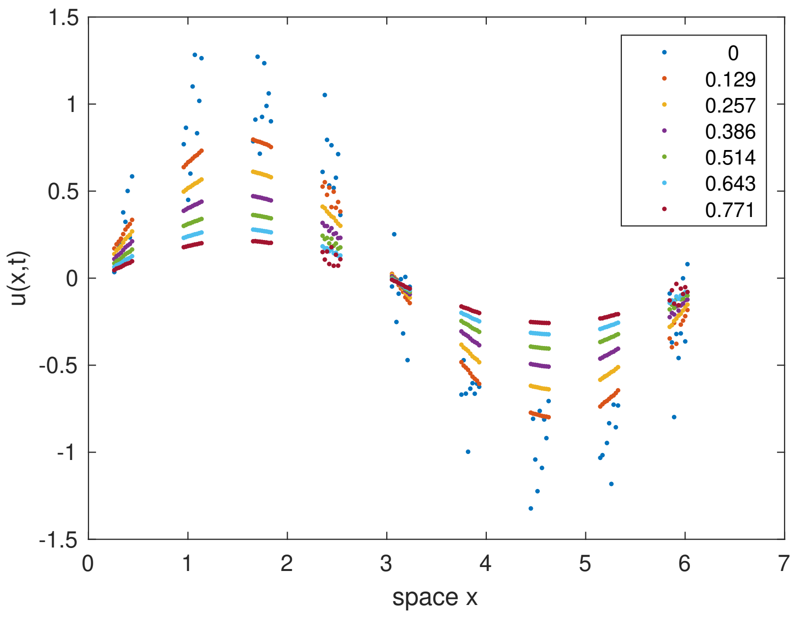


Figure 4.7: field $u(x,t)$ shows basic projective integration of patches of heterogeneous diffusion with ensemble average: different colours correspond to the times in the legend. Once transients have decayed, this field solution is smooth due to the ensemble average.



PIRK2 (Section 3.1), of bursts of simulation from heteroBurst (Section 4.5.3), as illustrated by Figures 4.6 and 4.7.

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

Plot the macroscale predictions to draw Figure 4.6 or Figure 4.7. If we have calculated an ensemble of field solutions, then we must first take the ensemble average.

```
if patches.EnsAve % calculate the ensemble average
  usAve=mean(reshape(us,size(us,1),length(xs(:)),nVars),3);
  ussAve=mean(reshape(uss,length(tss),length(xs(:)),nVars),3);
else
  usAve=us;
  ussAve=uss;
end
figure(2),clf
plot(xs(:),usAve','.')
ylabel('u(x,t)'), xlabel('space x')
legend(num2str(ts',3))
set(gcf,'PaperUnits','centimeters');
set(gcf,'PaperPosition',[0 0 14 10]);
if patches.EnsAve
  print('-depsc2','ps1HomogenisationUEnsAve')
else
  print('-depsc2','ps1HomogenisationU')
end
```

Also plot a surface detailing the microscale bursts as shown in Figure 4.8 or Figure 4.9.

Figure 4.8: stereo pair of the field $u(x, t)$ during each of the microscale bursts used in the projective integration with no ensemble averaging.

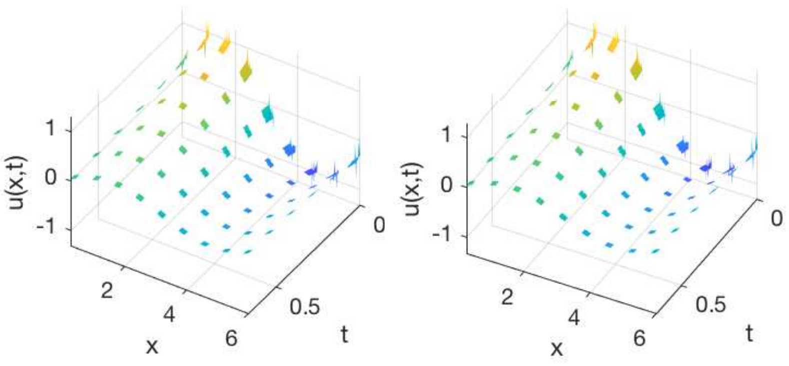
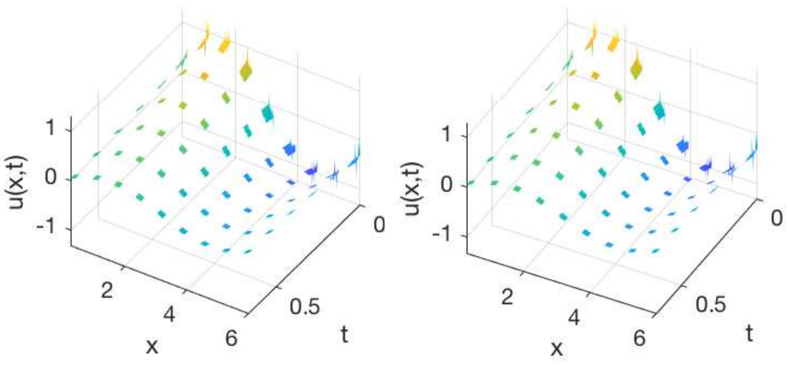


Figure 4.9: stereo pair of the field $u(x, t)$ during each of the microscale bursts used in the projective integration with ensemble averaging.



```
figure(3),clf
for k = 1:2, subplot(1,2,k)
  surf(tss,xs(:),ussAve',  'EdgeColor','none')
  ylabel('x'), xlabel('t'), zlabel('u(x,t)')
  axis tight, view(126-4*k,45)
end
set(gcf,'PaperUnits','centimeters');
set(gcf,'PaperPosition',[0 0 14 6]);
if patches.EnsAve
  print('-depsc2','ps1HomogenisationMicroEnsAve')
else
  print('-depsc2','ps1HomogenisationMicro')
end
```

End of the script.

4.5.2 `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 4.2), computes the time derivative (4.1) 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.cDiff.*diff(u))/dx^2; %- abs(u(i,:,:)).*u(i,:,:).
end% function
```

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

This code integrates in time the derivatives computed by `heteroDiff` from within the patch coupling of `patchSmooth1`. Try four possibilities:

- `ode23` generates 'noise' that is unsightly at best and may be ruinous;
- `ode45` is similar to `ode23`, but with reduced noise;
- `ode15s` does not cater for the NaNs in some components of `u`;
- `rk2int` simple specified step integrator, but may require inefficiently small time steps.

```
function [ts, ucts] = heteroBurst(ti, ui, bT)
  switch '45'
  case '23',  [ts,ucts] = ode23(@patchSmooth1,[ti ti+bT],ui(:));
  case '45',  [ts,ucts] = ode45(@patchSmooth1,[ti ti+bT],ui(:));
  case '15s', [ts,ucts] = ode15s(@patchSmooth1,[ti ti+bT],ui(:));
  case 'rk2', ts = linspace(ti,ti+bT,200)';
              ucts = rk2int(@patchSmooth1,ts,ui(:));
  end
end
```

Fin.

4.6 `waterWaveExample`: simulate a water wave PDE on patches

?? shows an example simulation in time generated by the patch scheme function applied to a simple wave PDE. The inter-patch coupling is realised by spectral interpolation to the patch edges of the mid-patch values.

This approach, based upon the differential equations coded in ??, may be adapted by a user to a wide variety of 1D wave and near-wave systems. For example, the differential equations of ?? describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (??).

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

Subsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. ?? 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 ?? 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 equi-spaced over the interior of the rectangle [`Xlim(1)`, `Xlim(2)`]$\times$ [`Xlim (3)`, `Xlim(4)`]: if of length two, then use the same interval in both directions, otherwise `Xlim(1:4)` give the interval in each direction.
- `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.

4.8 `patchSmooth2()`: interface to time integrators

Subsection contents

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)` $\cdot$ `prod(nPatch)` $\cdot$ `nVars` where there are `nVars` field values at each of the points in the `nSubP(1)`$\times$`nSubP(2)`$\times$ `nPatch(1)` $\times$ `nPatch(2)` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.

  - `.fun` is the name of the user's function `fun(t,u,x,y)` that computes the time derivatives on the patchy lattice. The array `u` has size `nSubP(1)` × `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.

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

Subsection 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` $\times$ `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` $\times$ `ny` $\times$ `Nx` $\times$ `Ny` $\times$ `nVars` array of the fields with edge values set by interpolation.

4.10 `wave2D`: example of a wave on patches in 2D

For $u(x, y, t)$, test and simulate the simple wave PDE in 2D space:

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u\,.$$

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

1. configPatches2
2. ode15s integrator $\leftrightarrow$ patchSmooth2 $\leftrightarrow$ wavePDE
3. process results

4.11 To do

- Testing is so far only qualitative. Need to be quantitative.
- Multiple space dimensions.
- Heterogeneous microscale via averaging regions.
- Parallel processing versions.
- ??
- Adapt to maps in micro-time? Surely easy, just an example.

4.12 Miscellaneous tests

4.12.1 `patchEdgeInt1test`: test the spectral interpolation

Subsection contents

A script to test the spectral interpolation of function `patchEdgeInt1()` Establish global data struct for the range of various cases.

```
clear all
global patches
nSubP=3
i0=(nSubP+1)/2; % centre-patch index
```

Test standard spectral interpolation Test over various numbers of patches, random domain lengths and random ratios.

```
for nPatch=5:10
nPatch=nPatch
Len=10*rand
ratio=0.5*rand
configPatches1(@sin,[0,Len],nan,nPatch,0,ratio,nSubP);
kMax=floor((nPatch-1)/2);
```

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

```
for k=-kMax:kMax
  u0=exp(1i*k*patches.x*2*pi/Len);
  ui=patchEdgeInt1(u0(:));
  normError=norm(ui-u0);
  if abs(normError)>5e-14
    normError=normError
    error(['failed single var interpolation k=' num2str(k)])
  end
end
```

Test multiple fields Set a profile, and evaluate the interpolation. For the case of the highest wavenumber, squash the error when the centre-patch values are all zero.

```
for k=1:nPatch/2
  u0=sin(k*patches.x*2*pi/Len);
  v0=cos(k*patches.x*2*pi/Len);
  uvi=patchEdgeInt1([u0(:);v0(:)]);
  normuError=norm(uvi(:,:,1)-u0)*norm(u0(i0,:));
  normvError=norm(uvi(:,:,2)-v0)*norm(v0(i0,:));
  if abs(normuError)+abs(normvError)>2e-13
    normuError=normuError, normvError=normvError
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End the for-loop over various geometries.

```
end
```

Now test spectral interpolation on staggered grid Must have even number of patches for a staggered grid.

```
for nPatch=6:2:20
nPatch=nPatch
ratio=0.5*rand
nSubP=3; % of form 4*N-1
Len=10*rand
configPatches1(@simpleWavepde,[0,Len],nan,nPatch,-1,ratio,nSubP);
kMax=floor((nPatch/2-1)/2)
```

Identify which microscale grid points are h or u values.

```
uPts=mod( bsxfun(@plus,(1:nSubP)',(1:nPatch)) ,2);
hPts=find(1-uPts);
uPts=find(uPts);
```

Set a profile for various wavenumbers. The capital letter `U` denotes an array of values merged from both u and h fields on the staggered grids.

```
fprintf('Single field-pair test.\n')
for k=-kMax:kMax
  U0=nan(nSubP,nPatch);
  U0(hPts)=rand*exp(+1i*k*patches.x(hPts)*2*pi/Len);
  U0(uPts)=rand*exp(-1i*k*patches.x(uPts)*2*pi/Len);
  Ui=patchEdgeInt1(U0(:));
  normError=norm(Ui-U0);
  if abs(normError)>5e-14
    normError=normError
    error(['failed single sys interpolation k=' num2str(k)])
  end
end
```

Test multiple fields Set a profile, and evaluate the interpolation. For the case of the highest wavenumber zig-zag, squash the error when the alternate centre-patch values are all zero. First shift the x-coordinates so that the zig-zag mode is centred on a patch.

```
fprintf('Two field-pairs test.\n')
x0=patches.x((nSubP+1)/2,1);
patches.x=patches.x-x0;
for k=1:nPatch/4
  U0=nan(nSubP,nPatch); V0=U0;
  U0(hPts)=rand*sin(k*patches.x(hPts)*2*pi/Len);
  U0(uPts)=rand*sin(k*patches.x(uPts)*2*pi/Len);
  V0(hPts)=rand*cos(k*patches.x(hPts)*2*pi/Len);
  V0(uPts)=rand*cos(k*patches.x(uPts)*2*pi/Len);
  UVi=patchEdgeInt1([U0(:);V0(:)]);
  normuError=norm(UVi(:,1:2:nPatch,1)-U0(:,1:2:nPatch))*norm(U0(i0,2:2:nPatch
      +norm(UVi(:,2:2:nPatch,1)-U0(:,2:2:nPatch))*norm(U0(i0,1:2:nPatch));
  normuError=norm(UVi(:,1:2:nPatch,2)-V0(:,1:2:nPatch))*norm(V0(i0,2:2:nPatch
      +norm(UVi(:,2:2:nPatch,2)-V0(:,2:2:nPatch))*norm(V0(i0,1:2:nPatch));
  if abs(normuError)+abs(normvError)>2e-13
    normuError=normuError, normvError=normvError
    error(['failed double field interpolation k=' num2str(k)])
  end
end
```

End for-loop over patches

```
end
```

Finish If no error messages, then all OK.

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

4.13 `patchEdgeInt2test`: tests 2D spectral interpolation

Try 99 realisations of random tests.

```
clear all, close all
global patches
for realisation=1:99
```

Choose and configure random sized domains, random sub-patch resolution, random size-ratios, random number of periodic-patches.

```
Lx=1+3*rand, Ly=1+3*rand
nSubP=1+2*randi(3,1,2)
ratios=rand(1,2)/2
nPatch=2+randi(4,1,2)
configPatches2(@sin,[0 Lx 0 Ly],nan,nPatch,0,ratios,nSubP)
```

Choose a random number of fields, then generate trigonometric shape with random wavenumber and random phase shift.

```
nV=randi(3)
[nx,Nx]=size(patches.x);
[ny,Ny]=size(patches.y);
u0s=nan(nx,ny,Nx,Ny,nV);
for iV=1:nV
  kx=randi([0 ceil((nPatch(1)-1)/2)])
  ky=randi([0 ceil((nPatch(2)-1)/2)])
  phix=pi*rand*(2*kx~=nPatch(1))
  phiy=pi*rand*(2*ky~=nPatch(2))
  % generate 2D array via auto-replication
  u0=sin(2*pi*kx*patches.x(:) /Lx+phix) ...
   .*sin(2*pi*ky*patches.y(:)'/Ly+phiy);
  % reshape into 4D array
  u0=reshape(u0,[nx Nx ny Ny]);
  u0=permute(u0,[1 3 2 4]);
  % store into 5D array
  u0s(:,:,:,:,iV)=u0;
end
```

Copy and NaN the edges, then interpolate

```
u=u0s; u([1 end],:,:,:,:)=nan; u(:,[1 end],:,:,:)=nan;
u=patchEdgeInt2(u(:));
```

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

```
err=u-u0s;
normerr=norm(err(:))
if normerr>1e-12, error('2D interpolation failed'), end
end
```

Bibliography

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

Calderon, C. P. (2007), 'Local diffusion models for stochastic reacting systems: estimation issues in equation-free numerics', *Molecular Simulation* **33**(9—10), 713—731.

Gear, C. W. & Kevrekidis, I. G. (2003*a*), '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*b*), '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.

Gustafsson, B. (1975), 'The convergence rate for difference approximations to mixed initial boundary value problems', *Mathematics of Computation* **29**(10), 396–406.

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

Plimpton, S., Thompson, A., Shan, R., Moore, S., Kohlmeyer, A., Crozier, P. & Stevens, M. (2016), Large-scale atomic/molecular massively parallel simulator, Technical report, `http://lammps.sandia.gov`.

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

Roberts, A. J. & Li, Z. (2006), 'An accurate and comprehensive model of thin fluid flows with inertia on curved substrates', *J. Fluid Mech.* **553**, 33–73.

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

Samaey, G., Roberts, A. J. & Kevrekidis, I. G. (2010), Equation-free computation: an overview of patch dynamics, *in* J. Fish, ed., 'Multiscale methods: bridging the scales in science and engineering', Oxford University Press, chapter 8, pp. 216–246.

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

Svard, M. & Nordstrom, J. (2006), 'On the order of accuracy for difference approximations of initial-boundary value problems', *Journal of Computational Physics* **218**, 333–352.