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

This Developers Manual contains line-by-line descriptions of the code in each function in the toolbox, and each example. For basic descriptions of each function, quick start guides, and some basic examples, see the User Manual.

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. Appendix A 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.

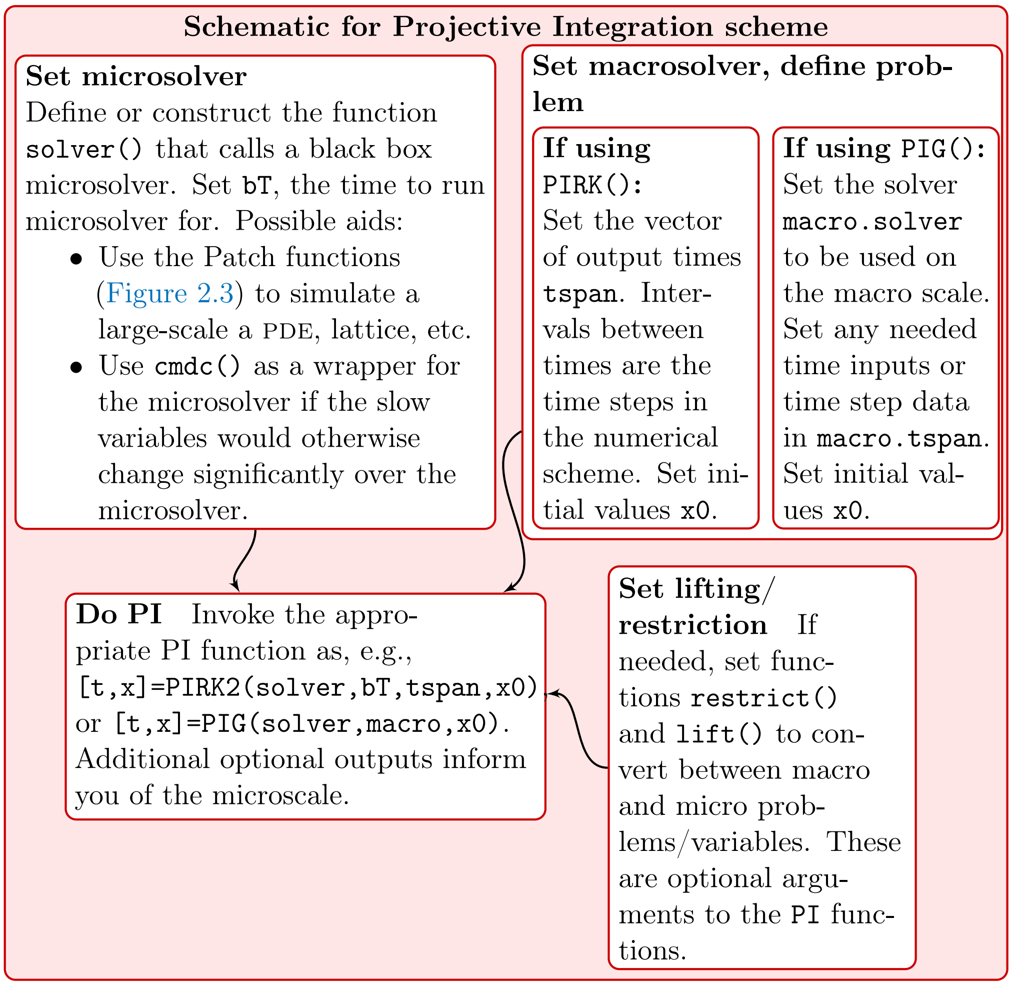
The cheat sheet consists of two flow charts. Figure 2.1 overviews constructing a PI simulation. Figure 2.2 roughly guides which of the top-level PI functions should be used.

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

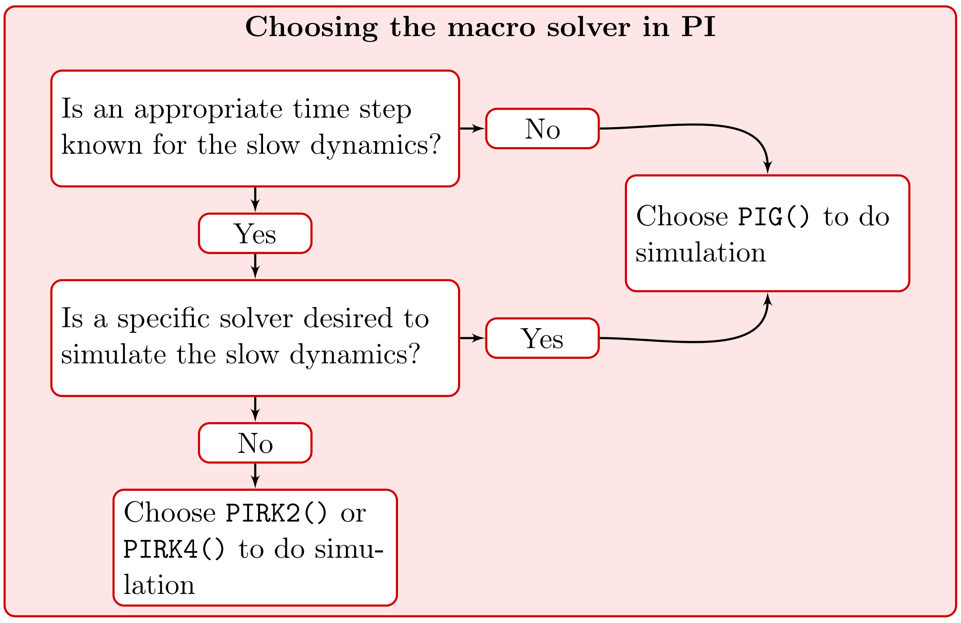
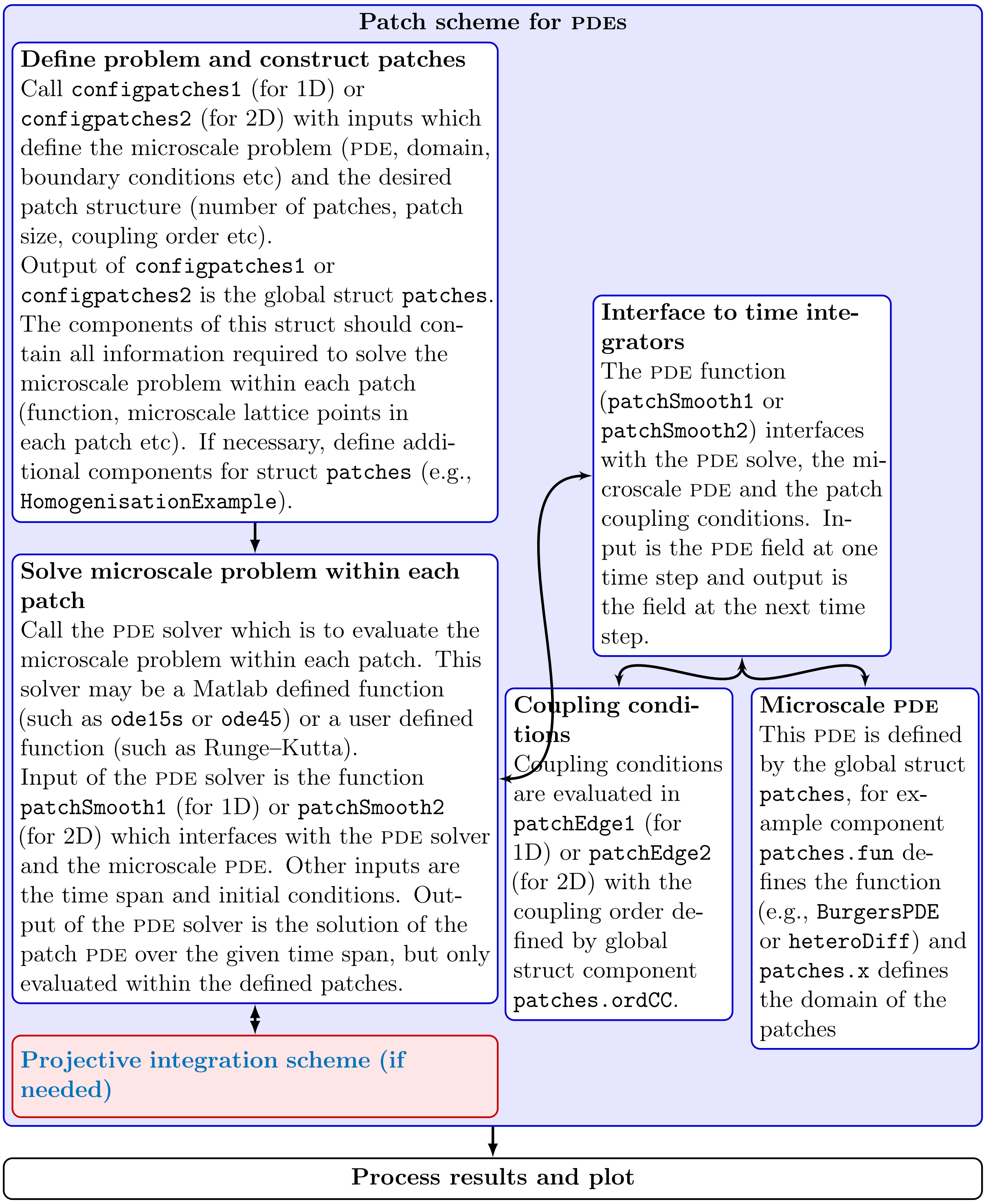

Choosing the macro solver in PI
Is an appropriate time step known for the slow dynamics?
No
Yes
Choose PIG() to do simulation
Is a specific solver desired to simulate the slow dynamics?
Yes
No
Choose PIRK2() or PIRK4() to do simulation

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.

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.

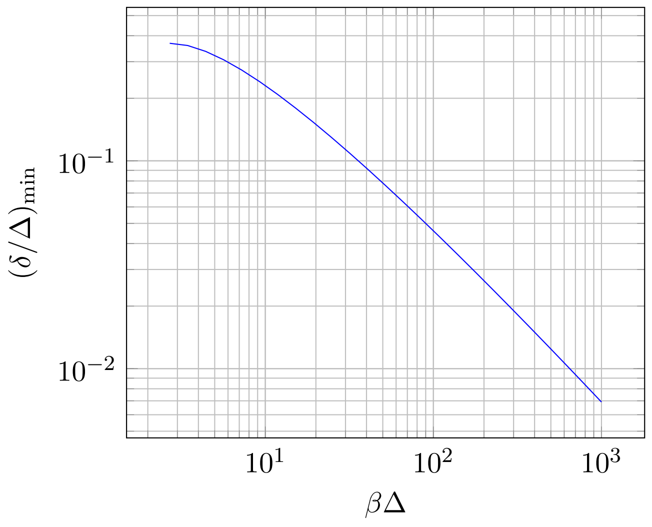
  – Outputs: `tOut`, the column vector of solution times; and `xOut`, an array in which each *row* contains the system state at corresponding times.

- `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).

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.



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.

- `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
```

3.1.2 The projective integration code

Determine the number of time steps and preallocate storage for macroscale estimates.

```
nT=length(tSpan);
x=nan(nT,length(x0));
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
nArgs=nargout();
saveMicro = (nArgs>1);
saveFullMicro = (nArgs>3);
saveSvf = (nArgs>4);
```

Run a preliminary application of the microBurst on the initial conditions to help relax to the slow manifold. This is done in addition to the microBurst in the main loop, because the initial conditions are often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```
x0 = reshape(x0,1,[]);
[relax_t,relax_x0] = microBurst(tSpan(1),x0,bT);
```

Use the end point of the microBurst as the initial conditions.

```
tSpan(1) = tSpan(1)+bT;
x(1,:)=relax_x0(end,:);
```

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

```
if saveMicro
    tms = cell(nT,1);
    xms = cell(nT,1);
    tms{1} = reshape(relax_t,[],1);
    xms{1} = relax_x0;
    if saveFullMicro
        rm.t = cell(nT,1);
        rm.x = cell(nT,1);
        if saveSvf
            svf.t = nan(2*nT-2,1);
            svf.dx = nan(2*nT-2,length(x0));
        end
    end
end
```

Loop over the macroscale time steps

```
for jT = 2:nT
    T = tSpan(jT-1);
```

If two applications of the microBurst would cover one entire macroscale time-step, then do so (setting some internal states to NaN); else proceed to projective step.

```
    if 2*abs(bT)>=abs(tSpan(jT)-T) & bT*(tSpan(jT)-T)>0
        [t1,xm1] = microBurst(T, x(jT-1,:), tSpan(jT)-T);
```

```
      x(jT,:) = xm1(end,:);
      t2=nan; xm2=nan(1,size(xm1,2));
      dx1=xm2; dx2=xm2;
    else
```

Run the first application of the microBurst; since this application directly follows from the initial conditions, or from the latest macrostep, this microscale information is physically meaningful as a simulation of the system. Extract the size of the final time step.

```
    [t1,xm1] = microBurst(T, x(jT-1,:), bT);
    del = t1(end)-t1(end-1);
```

Check for round-off error.

```
    xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
    roundingTol=1e-8;
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 1st projection at T=' num2str(T)
    end
```

Find the needed time step to reach time `tSpan(n+1)` and form a first estimate `dx1` of the slow vector field.

```
    Dt = tSpan(jT)-T-bT;
    dx1 = (xm1(end,:)-xm1(end-1,:))/del;
```

Project along `dx1` to form an intermediate approximation of `x`; run another application of the microBurst and form a second estimate of the slow vector field.

```
    xint = xm1(end,:) + (Dt-bT)*dx1;
    [t2,xm2] = microBurst(T+Dt, xint, bT);
    del = t2(end)-t2(end-1);
    dx2 = (xm2(end,:)-xm2(end-1,:))/del;
```

Check for round-off error.

```
    xt=[reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 2nd projection at T=' num2str(T)
    end
```

Use the weighted average of the estimates of the slow vector field to take a macrostep.

```
    x(jT,:) = xm1(end,:) + Dt*(dx1+dx2)/2;
```

Now end the if-statement that tests whether a projective step saves simulation time.

```
    end
```

If saving trusted microscale data, then populate the cell arrays for the current loop iterate with the time steps and output of the first application of the microBurst. Separate bursts by `NaN`s.

```
    if saveMicro
        tms{jT} = [reshape(t1,[],1); nan];
        xms{jT} = [xm1; nan(1,size(xm1,2))];
```

If saving all microscale data, then repeat for the remaining applications of the microBurst.

```
        if saveFullMicro
            rm.t{jT} = [reshape(t2,[],1); nan];
            rm.x{jT} = [xm2; nan(1,size(xm2,2))];
```

If saving Projective Integration estimates of the slow vector field, then populate the corresponding cells with times and estimates.

```
            if saveSvf
                svf.t(2*jT-3:2*jT-2) = [t1(end); t2(end)];
                svf.dx(2*jT-3:2*jT-2,:) = [dx1; dx2];
            end
        end
    end
```

Terminate the main loop:

```
end
```

Overwrite `x(1,:)` with the specified initial condition `tSpan(1)`.

```
x(1,:) = reshape(x0,1,[]);
```

For additional requested output, concatenate all the cells of time and state data into two arrays.

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveFullMicro
        rm.t = cell2mat(rm.t);
        rm.x = cell2mat(rm.x);
    end
end
```

3.1.3 If no output specified, then plot simulation

```
if nArgs==0
    figure, plot(tSpan,x,'o:')
    title('Projective Simulation with PIRK2')
end
```

This concludes `PIRK2()`.

```
end
```

3.2 `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.

3.2.1 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
```

Find the number of time steps at which output is expected, and the number of variables.

```
nT=length(tSpan)-1;
nx = length(lift(x0));
```

Get the number of expected outputs and set logical indices to flag what data should be saved. If no lifting/restriction operators were set, assign them.

```
nArgs=nargout();
saveMicro = (nArgs>1);
saveSvf = (nArgs>2);
if nargin < 5 %no lift/restrict operators
    lift=@(x) x;
    restrict=@(x) x;
end
```

Run a first application of the microBurst on the initial conditions. This is done in addition to the microBurst in the main loop, because the initial conditions are often far from the attracting slow manifold.

```
x0 = reshape(x0,[],1);
[relax_t,x0_micro_relax] = microBurst(tSpan(1),lift(x0));
x0_relax = restrict(x0_micro_relax);
```

Update the initial time.

```
tSpan(1) = relax_t(end);
```

Allocate cell arrays for times and states for any of the outputs requested by the user. If saving information, then record the first application of the microBurst. Note that it is unknown a priori how many applications of the microBurst will be required; this code may be run more efficiently if the correct number is used in place of `nT+1` as the dimension of the cell arrays.

```
if saveMicro
    tms=cell(nT+1,1); xms=cell(nT+1,1);
    n=1;
    tms{n} = reshape(relax_t,[],1);
    xms{n} = x0_micro_relax;

    if saveSvf
        svf.t = cell(nT+1,1);
        svf.dx = cell(nT+1,1);
    end
end
```

The idea of `PIG()` is to use the output from the microBurst to approximate an unknown function `ff(t,x)`, that describes the slow dynamics. This approximation is then used in the system/user-defined ‘coarse solver’ `macroInt()`. The approximation is described in

```
function [dx]=genProjection(tt,xx)
```

Run a microBurst from the given initial conditions.

```
    [t_tmp,x_micro_tmp] = microBurst(tt,reshape(lift(xx),[],1));
```

Compute the standard Projective Integration approximation of the slow vector field.

```
        del = t_tmp(end)-t_tmp(end-1);
        dx = ( restrict(x_micro_tmp(end,:))-restrict(x_micro_tmp(end-1,:)) )'
```

Save the microscale data, and the Projective Integration slow vector field, if requested.

```
        if saveMicro
        n=n+1;
        tms{n} = [reshape(t_tmp,[],1); nan];
        xms{n} = [x_micro_tmp; nan(1,nx)];
        if saveSvf
            svf.t{n-1} = tt;
            svf.dx{n-1} = dx;
        end
        end
end% function genProjection()
```

Define the approximate slow vector field according to Projective Integration.

```
ff=@(t,x) genProjection(t,x);
```

Do Projective Integration of `ff()` with the user-specified microBurst.

```
[t,x]=feval(macroInt,ff,tSpan,x0_relax(end,:)');
```

Overwrite `x(1,:)` and `t(1)`, which the user expect to be `x0` and `tSpan(1)` respectively, with the given initial conditions.

```
x(1,:) = x0';
t(1) = tSpan(1);
```

For each additional requested output, concatenate all the cells of time and state data into two arrays. Then, return the two arrays in a cell.

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveSvf
        svf.t = cell2mat(svf.t);
        svf.dx = cell2mat(svf.dx);
    end
end
```

3.2.2 If no output specified, then plot simulation

```
if nArgs==0
    fifure, plot(t,x,'o:')
    title('Projective Simulation via PIG')
end
```

This concludes `PIG()`.

```
end
```

3.3 `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 $-5 \leq t \leq 0$ 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 $\Delta = -1$.

```
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
```

Input

- `solver()`, a function that produces output from the user-specified code for microscale simulation.

  `[tOut, xOut] = solver(tStart, xStart, tSim)`

  - Inputs: `tStart`, the start time of a burst of simulation; `xStart`, the row n-vector of the starting state; `tSim`, 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 microsolver 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. `PIRK4()` does not use adaptive time stepping; the macroscale time steps are (nearly) the steps between elements of `tSpan`.

- `x0` is an n-vector of initial values at the initial time `tSpan(1)`. Elements of `x0` may be `NaN`: they are included in the simulation and output, and often represent boundaries in space fields.

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 = PIRK4(solver,bT,tSpan,x0)`.

  However, microscale details of the underlying Projective Integration computations may be helpful. `PIRK4()` 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 microsolver 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 4ℓ 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 $4\ell \times n$ array containing the estimated slow vector field.

3.3.1 The projective integration code

Determine the number of time steps and preallocate storage for macroscale estimates.

```
nT=length(tSpan);
x=nan(nT,length(x0));
```

Get the number of expected outputs and set logical indices to flag what data should be saved.

```
nArgs=nargout();
saveMicro = (nArgs>1);
saveFullMicro = (nArgs>3);
saveSvf = (nArgs>4);
```

Run a preliminary application of the microsolver on the initial conditions to help relax to the slow manifold. This is done in addition to the microsolver in the main loop, because the initial conditions are often far from the attracting slow manifold. Require the user to input and output rows of the system state.

```
x0 = reshape(x0,1,[]);
[relax_t,relax_x0] = solver(tSpan(1),x0,bT);
```

Use the end point of the microsolver as the initial conditions.

```
tSpan(1) = tSpan(1)+bT;
x(1,:)=relax_x0(end,:);
```

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

```
if saveMicro
    tms = cell(nT,1);
    xms = cell(nT,1);
    tms{1} = reshape(relax_t,[],1);
    xms{1} = relax_x0;
    if saveFullMicro
        rm.t = cell(nT,1);
        rm.x = cell(nT,1);
        if saveSvf
            svf.t = nan(4*nT-4,1);
            svf.dx = nan(4*nT-4,length(x0));
        end
    end
end
```

Loop over the macroscale time steps

```
for jT = 2:nT
    T = tSpan(jT-1);
```

If four applications of the microsolver would cover the entire macroscale time-step, then do so (setting some internal states to `NaN`); else proceed to projective step.

```
    if 4*abs(bT)>=abs(tSpan(jT)-T) & bT*(tSpan(jT)-T)>0
        [t1,xm1] = solver(T, x(jT-1,:), tSpan(jT)-T);
        x(jT,:) = xm1(end,:);
        t2=nan; xm2=nan(1,size(xm1,2));
        t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
    else
```

Run the first application of the microsolver; since this application directly follows from the initial conditions, or from the latest macrostep, this microscale information is physically meaningful as a simulation of the system. Extract the size of the final time step.

```
    [t1,xm1] = solver(T, x(jT-1,:), bT);
    del = t1(end)-t1(end-1);
```

Check for round-off error.

```
    xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
    roundingTol=1e-8;
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 1st projection at T=' num2str(T)
    end
```

Find the needed time step to reach time `tSpan(n+1)` and form a first estimate `dx1` of the slow vector field.

```
    Dt = tSpan(jT)-T-bT;
    dx1 = (xm1(end,:)-xm1(end-1,:))/del;
```

Project along `dx1` to form an intermediate approximation of `x`; run another application of the microsolver and form a second estimate of the slow vector field.

```
    xint = xm1(end,:) + (Dt/2-bT)*dx1;
    [t2,xm2] = solver(T+Dt/2, xint, bT);
    del = t2(end)-t2(end-1);
    dx2 = (xm2(end,:)-xm2(end-1,:))/del;

    xint = xm1(end,:) + (Dt/2-bT)*dx2;
    [t3,xm3] = solver(T+Dt/2, xint, bT);
    del = t3(end)-t3(end-1);
    dx3 = (xm3(end,:)-xm3(end-1,:))/del;

    xint = xm1(end,:) + (Dt-bT)*dx3;
    [t4,xm4] = solver(T+Dt, xint, bT);
    del = t4(end)-t4(end-1);
    dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

```
    xt=[reshape(t2(end-1:end),[],1) xm2(end-1:end,:)];
    if norm(diff(xt))/norm(xt,'fro') < roundingTol
    warning(['significant round-off error in 2nd projection at T=' num2str(T)
    end
```

Use the weighted average of the estimates of the slow vector field to take a macrostep.

```
    x(jT,:) = xm1(end,:) + Dt*(dx1 + 2*dx2 + 2*dx3 + dx4)/6;
```

Now end the if-statement that tests whether a projective step saves simulation time.

```
    end
```

If saving trusted microscale data, then populate the cell arrays for the current loop iterate with the time steps and output of the first application of the microsolver. Separate bursts by `NaN`s.

```
    if saveMicro
        tms{jT} = [reshape(t1,[],1); nan];
        xms{jT} = [xm1; nan(1,size(xm1,2))];
```

If saving all microscale data, then repeat for the remaining applications of the microsolver.

```
        if saveFullMicro
            rm.t{jT} = [reshape(t2,[],1); nan;...
                        reshape(t3,[],1); nan;...
                        reshape(t4,[],1); nan];
            rm.x{jT} = [xm2; nan(1,size(xm2,2));...
                        xm3; nan(1,size(xm2,2));...
                        xm4; nan(1,size(xm2,2))];
```

If saving Projective Integration estimates of the slow vector field, then populate the corresponding cells with times and estimates.

```
            if saveSvf
                svf.t(4*jT-7:4*jT-4) = [t1(end); t2(end); t3(end); t4(end)];
                svf.dx(4*jT-7:4*jT-4,:) = [dx1; dx2; dx3; dx4];
            end
        end
    end
```

Terminate the main loop:

```
end
```

Overwrite `x(1,:)` with the specified initial condition `tSpan(1)`.

```
x(1,:) = reshape(x0,1,[]);
```

For additional requested output, concatenate all the cells of time and state data into two arrays.

```
if saveMicro
    tms = cell2mat(tms);
    xms = cell2mat(xms);
    if saveFullMicro
        rm.t = cell2mat(rm.t);
        rm.x = cell2mat(rm.x);
    end
end
```

3.3.2 If no output specified, then plot simulation

```
if nArgs==0
    figure, plot(tSpan,x,'o:')
    title('Projective Simulation with PIRK4')
end
```

This concludes `PIRK4()`.

```
end
```

3.3.3 `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.

Begin with a standard application of the micro-burst.

```
[ts,xs] = feval(microBurst,t0,x0);
bT = ts(end)-ts(1);
```

Project backwards to before the initial time, then simulate just one burst forward to obtain a simulation burst that ends at the original `t0`.

```
dxdt = (xs(end,:) - xs(end-1,:))/(ts(end,:) - ts(end-1,:));
x0 = xs(end,:)-2*bT*dxdt;
t0 = ts(1)-bT;
[ts,xs] = feval(microBurst,t0,x0.');
```

3.4 Example: PI using Runge–Kutta macrosolvers

This script is a demonstration of the `PIRK()` schemes, that use a Runge–Kutta macrosolver, applied to a simple linear system with some slow and fast directions.

Clear workspace and set a seed.

```
clear
rng(1)
```

The majority of this example involves setting up details for the microsolver. We use a simple function `gen_linear_system()` that outputs a function $f(t,x) = \mathbf{A}\vec{x} + \vec{b}$, where $\mathbf{A}$ has some eigenvalues with large negative real part, corresponding to fast variables and some eigenvalues with real part close to zero, corresponding to slow variables. The function `gen_linear_system()` requires that we specify bounds on the real part of the strongly stable eigenvalues,

```
fastband = [-5e2; -1e2];
```

and bounds on the real part of the weakly stable/unstable eigenvalues,

```
slowband = [-0.002; 0.002];
```

We now generate a random linear system with seven fast and three slow variables.

```
f = gen_linear_system(7,3,fastband,slowband);
```

Set the time step size and total integration time of the microsolver.

```
dt = 0.001;
bT = 0.05;
```

As a rule of thumb, the time steps `dt` should satisfy `dt` $\leq 1/|$`fastband`$(1)|$ and the time to simulate with each application of the microsolver, `micro.bT`, should be larger than or equal to $1/|$`fastband`$(2)|$. We set the integration scheme to be used in the microsolver. Since the time steps are so small, we just use the forward Euler scheme

```
solver='fe';
```

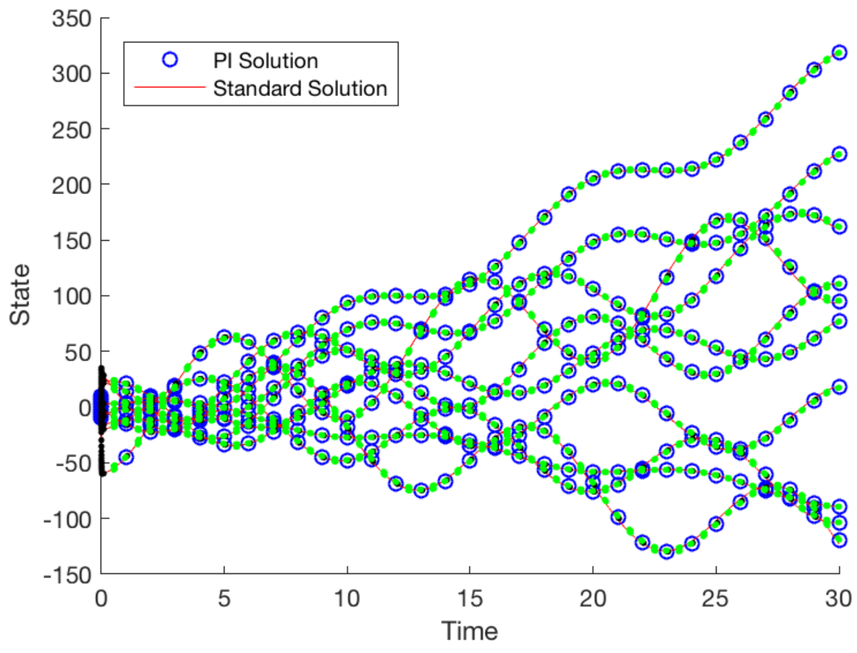
(Other options: `'rk2'` for second order Runge–Kutta, `'rk4'` for fourth order, or any Matlab/Octave integrator such as `'ode45'`.)

A crucial part of the PI philosophy is that it does not assume anything about the microsolver. For this reason, the microsolver must be a 'black box', which is run by specifying an initial time and state, and a duration to simulate for. All the details of the microsolver must be set by the user. We generate and save a black box microsolver.

```
bbm = bbgen(solver,f,dt);
solver = bbm;
```

Set the macroscale times at which we request output from the PI scheme and the initial conditions.

Figure 3.2: Demonstration of PIRK4(). From initial conditions, the system rapidly trannsitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.



```
tSpan=0: 1 : 30;
IC = linspace(-10,10,10);
```

We implement the PI scheme, saving the coarse states in `x`, the ‘trusted’ applications of the microsolver in `xmicro`, and the additional applications of the microsolver in `xrmicro`. Note that the second and third outputs are optional and do not need to be set.

```
[x, tms, xms, rm] = PIRK4(solver, bT, tSpan, IC);
```

For verification, we also compute the trajectories using a standard solver.

```
[tt,ode45x] = ode45(f,tSpan([1,end]),IC);
```

Figure 3.2 plots the output.

```
tmsr = rm.t; xmsr = rm.x;
clf()
hold on
PI_sol=plot(tSpan,x,'bo');
std_sol=plot(tt,ode45x,'r');
plot(tms,xms,'k.');
plot(tmsr,xmsr,'g.');
legend([PI_sol(1),std_sol(1)],'PI Solution',...
    'Standard Solution','Location','NorthWest')
xlabel('Time');
```

```
ylabel('State');
```

Save plot to a file.

```
set(gcf,'PaperPosition',[0 0 14 10])
print('-depsc2','PIRK')
```

3.5 Example: Projective Integration using General macrosolvers

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation. The aim is to use a standard non-stiff numerical integrator, such as `ode45()`, on the slow, long-time macroscale. For this stiff system, `PIG()` is an order of magnitude faster than ordinary use of `ode45`.

```
clear all, close all
```

Set time scale separation and model.

```
epsilon = 1e-4;
dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
              (cos(x(1))-x(2))/epsilon ];
```

Set the 'black box' microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

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

Set initial conditions, and the time to be covered by the macrosolver.

```
x0 = [1 1.4];
tSpan=[0 15];
```

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

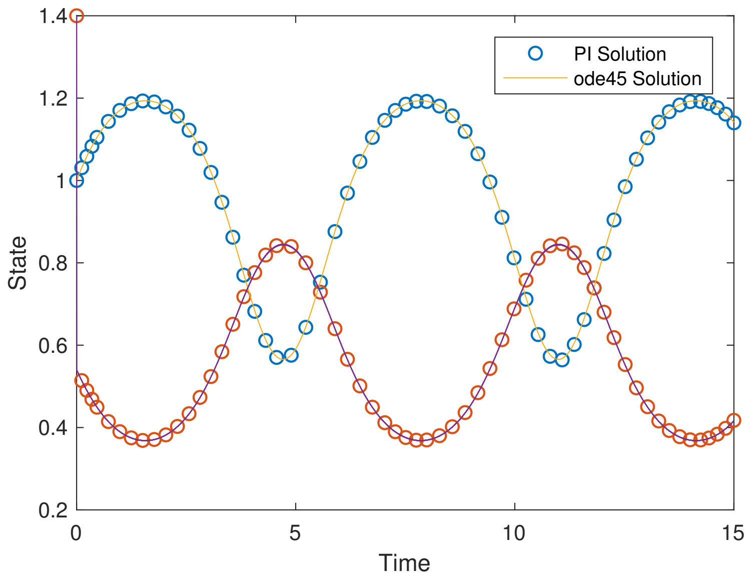
```
tic
[ts,xs,tms,xms] = PIG('ode45',microBurst,tSpan,x0);
tPIGusingODE45asMacro = toc
tic
[t45,x45] = ode45(dxdt,tSpan,x0);
tODE45alone = toc
```

Plot the output on two figures, showing the truth and macrosteps on both, and all applications of the microsolver on the first figure.

```
figure
h = plot(ts,xs,'o', t45,x45,'-', tms,xms,'.');
legend(h(1:2:5),'PI Solution','ode45 Solution','PI microsolver')
xlabel('Time'), ylabel('State')

figure
h = plot(ts,xs,'o', t45,x45,'-');
```

Figure 3.3: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver `ode45`.



```
legend(h([1 3]),'PI Solution','ode45 Solution')
xlabel('Time'), ylabel('State')
set(gcf,'PaperPosition',[0 0 14 10]), print('-depsc2','PIGExample')
```

Figure 3.3 plots the output.

- The problem may be made more, or less, stiff by changing the time-scale separation parameter $\epsilon =$ `epsilon`. The compute time of `PIG()` is almost independent of ϵ, whereas that of `ode45()` is proportional to $1/\epsilon$.

  But if the problem is insufficiently stiff (larger ϵ), then `PIG()` produces nonsense. This nonsense is overcome by `cdmc()` (Section 3.6).

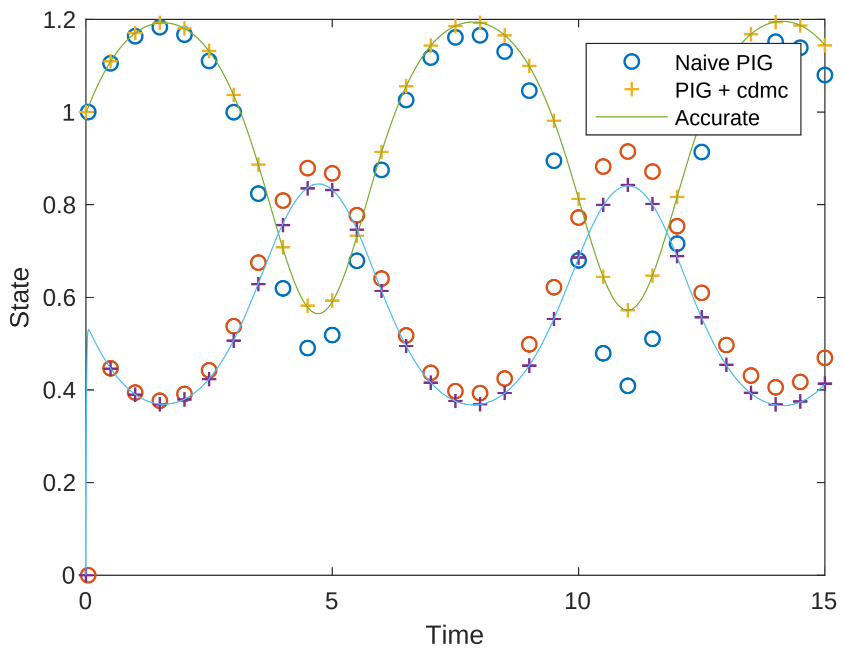
- The mildly stiff problem in Section 3.4 may be efficiently solved by a standard solver (e.g., `ode45()`). The stiff but low dimensional problem in this example can be solved efficiently by a standard stiff solver (e.g., `ode15s()`). The real advantage of the Projective Integration schemes is in high dimensional stiff problems, that cannot be efficiently solved by most standard methods.

3.6 Explore: Projective Integration using constraint-defined manifold computing

In this example the Projective Integration-General scheme is applied to a singularly perturbed ordinary differential equation in which the time scale separation is not large. The resulting simulation is not accurate. In parallel, we run the same scheme but with `cdmc()` used as a wrapper for the microsolver. This second implementation successfully replicates the true dynamics.

```
clear all, close all
```

Figure 3.4: Accurate simulation of a weakly stiff non-autonomous system by `PIG()` using cdmc(), and an inaccurate solution using a naive application of `PIG()`.



Set a weak time scale separation and model.

```
epsilon = 0.01;
dxdt=@(t,x) [ cos(x(1))*sin(x(2))*cos(t)
              (cos(x(1))-x(2))/epsilon ];
```

Set the 'naive' microsolver to be an integration using a standard solver, and set the standard time of simulation for the microsolver.

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

Create a second struct in which the solver is the output of `cdmc()`.

```
cBurst = @(t,x) cdmc(naiveBurst,t,x);
```

Set initial conditions, and the time to be covered by the macrosolver.

```
x0 = [1 0];
tSpan=0:0.5:15;
```

Simulate using `PIG()` with each of the above microsolvers. Generate a trusted solution using standard numerical methods.

```
[nt,nx] = PIG('ode45',naiveBurst,tSpan,x0);
[ct,cx] = PIG('ode45',cBurst,tSpan,x0);
[t45,x45] = ode45(dxdt,tSpan([1 end]),x0);
```

Figure 3.4 plots the output.

```
figure
```

```
h = plot(nt,nx,'o', ct,cx,'+', t45,x45,'-');
legend(h(1:2:5),'Naive PIG','PIG + cdmc','Accurate')
xlabel('Time'), ylabel('State')
set(gcf,'PaperPosition',[0 0 14 10]), print('-depsc2','PIGExplore')
```

The source of the error in the standard `PIG()` scheme is the burst length `bT`, that is significant on the slow time scale. Set `bT` to `20*epsilon` or `50*epsilon`[1] to worsen the error in both schemes. This example reflects a general principle, that most Projective Integration schemes will incur a global error term which is proportional to the simulation time of the microsolver and independent of the order of the microsolver. The `PIRK()` schemes have been written to minimise, if not eliminate entirely, this error, but by design `PIG()` works with any user-defined macrosolver and cannot reduce this error. The function `cdmc()` reduces this error term by attempting to mimic the microsolver without advancing time.

3.7 To do/discuss

- could implement Projective Integration by 'arbitrary' Runge–Kutta scheme; that is, by having the user input a particular Butcher table—surely only specialists would be interested
- can 'reverse' the order of projection and microsolver applications with a little fiddling. Then output at each user-requested coarse time is the end point of an application of the microsolver - better predictions for fast variables.
- Can maybe implement microsolvers that terminate a burst when the fast dynamics have settled using, for example, the 'Events' function handle in ode23.

[1] this example is quite extreme: at bT=50*epsilon, it would be computationally much cheaper to simulate the entire length of tSpan using the microsolver alone.

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 Sections 4.1.1 and 4.4.1 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()`. Section 4.1.1 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 4.1.1 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.

4.1.1 If no arguments, then execute an example

```
if nargin==0
```

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

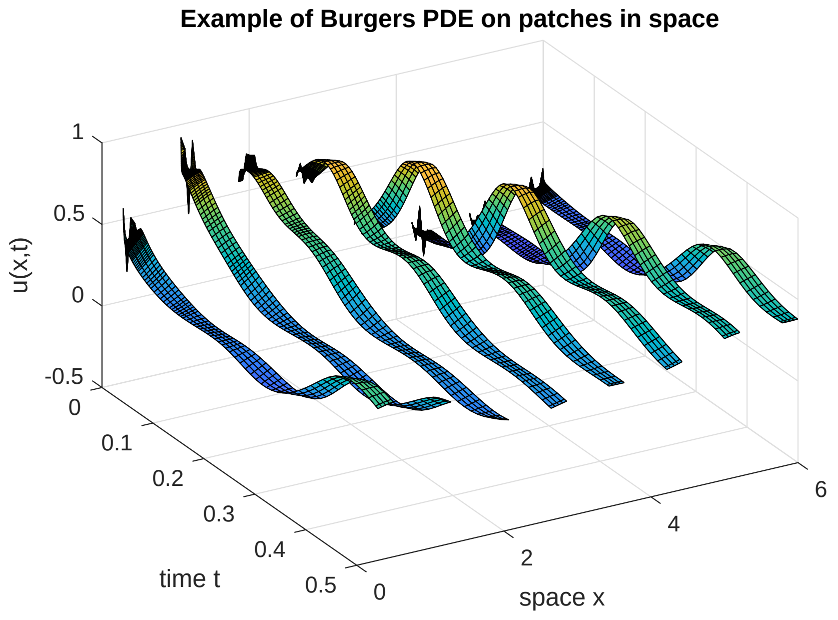
1. configPatches1
2. ode15s integrator ↔ patchSmooth1 ↔ user's burgersPDE
3. process results

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 points within each patch.

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

Set an initial condition, and integrate in time using standard functions.

Figure 4.1: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.



```
u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
[ts,ucts]=ode15s(@patchSmooth1,[0 0.5],u0(:));
```

Plot the simulation using only the microscale values interior to the patches: set x-edges to `nan` to leave the gaps. Figure 4.1 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 $u_t = u_{xx} - 30uu_x$, 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
```

4.1.2 The code to make patches

Set one edge-value to compute by interpolation if not specified by the user. Store in the struct.

```
if nargin<8, nEdge=1; end
if nEdge>1, error('multi-edge-value interp not yet implemented'), end
if 2*nEdge+1>nSubP, error('too many edge values requested'), end
patches.nEdge=nEdge;
```

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

```
patches.fun=fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 and −1.

```
if (ordCC<-1) | ~(floor(ordCC)==ordCC)
    error('ordCC out of allowed range integer>-2')
end
```

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

```
patches.alt=mod(ordCC,2);
ordCC=ordCC+patches.alt;
patches.ordCC=ordCC;
```

Check for staggered grid and periodic case.

```
  if patches.alt & (mod(nPatch,2)==1)
    error('Require an even number of patches for staggered grid')
  end
```

Might as well precompute the weightings for the interpolation of field values for coupling. (Could sometime extend to coupling via derivative values.)

```
patches.Cwtsr=zeros(ordCC,1);
if patches.alt  % eqn (7) in \cite{Cao2014a}
    patches.Cwtsr(1:2:ordCC)=[1 ...
      cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
      factorial(2*(1:(ordCC/2-1)))];
    patches.Cwtsr(2:2:ordCC)=[ratio/2 ...
      cumprod((ratio^2-(1:2:(ordCC-2)).^2)/4)./ ...
      factorial(2*(1:(ordCC/2-1))+1)*ratio/2];
else %
    patches.Cwtsr(1:2:ordCC)=(cumprod(ratio^2- ...
      (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))-1)/ratio);
    patches.Cwtsr(2:2:ordCC)=(cumprod(ratio^2- ...
      (((1:(ordCC/2))-1)).^2)./factorial(2*(1:(ordCC/2))));
end
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

Third, set the centre of the patches in a the macroscale grid of patches assuming periodic macroscale domain.

```
X=linspace(Xlim(1),Xlim(2),nPatch+1);
X=X(1:nPatch)+diff(X)/2;
DX=X(2)-X(1);
```

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio` · `DX`.

```
if mod(nSubP,2)==0, error('configPatches1: nSubP must be odd'), end
i0=(nSubP+1)/2;
dx=ratio*DX/(i0-1);
patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
end% function
```

Fin.

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.

Reshape the fields `u` as a 2/3D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 4.3 describes `patchEdgeInt1()`.

```
u=patchEdgeInt1(u);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero, then return to a to the user/ integrator as column vector.

```
dudt=patches.fun(t,u,patches.x);
dudt([1 end],:,:)=0;
dudt=reshape(dudt,[],1);
```

Fin.

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

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

```
[nSubP,nPatch]=size(patches.x);
nVars=round(numel(u)/numel(patches.x));
if numel(u)~=nSubP*nPatch*nVars
  nSubP=nSubP, nPatch=nPatch, nVars=nVars, sizeu=size(u)
  end
u=reshape(u,nSubP,nPatch,nVars);
```

Compute lattice sizes from inside the patches as the edge values may be NaNs, etc.

```
dx=patches.x(3,1)-patches.x(2,1);
DX=patches.x(2,2)-patches.x(2,1);
```

If the user has not defined the patch core, then we assume it to be a single point in the middle of the patch. For `patches.nCore` $\neq 1$ the half width ratio is reduced, as described by **?**.

```
if ~isfield(patches,'nCore')
    patches.nCore=1;
end
r=dx*(nSubP-1)/2/DX*(nSubP - patches.nCore)/(nSubP - 1);
```

For the moment assume the physical domain is macroscale periodic so that the coupling formulas are simplest. Should eventually cater for periodic, odd-mid-gap, even-mid-gap, even-mid-patch, Dirichlet, Neumann etc. These index vectors point to patches and their two immediate neighbours.

```
j=1:nPatch; jp=mod(j,nPatch)+1; jm=mod(j-2,nPatch)+1;
```

Calculate centre of each patch and the surrounding core. (`nSubP` and `nCore` are both odd)

```
i0=round((nSubP+1)/2);
c=round((patches.nCore-1)/2);
```

Lagrange interpolation gives patch-edge values so compute centred differences of the patch core averages for the macro-interpolation of all fields. Assumes the domain is macro-periodic.

```
if patches.ordCC>0 % then non-spectral interpolation
  if patches.EnsAve
```

```
    ucore=sum(mean(u((i0-c):(i0+c),j,:),3),1)';
    dmu=zeros(patches.ordCC,nPatch);
  else
    ucore=reshape(sum(u((i0-c):(i0+c),j,:),1),nPatch,nVars);
    dmu=zeros(patches.ordCC,nPatch,nVars);
  end;
  if patches.alt % use only odd numbered neighbours
    dmu(1,:,:)=(ucore(jp,:)+ucore(jm,:))/2; % \mu
    dmu(2,:,:)=(ucore(jp,:)-ucore(jm,:)); % \delta
    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
  else % standard
    dmu(1,j,:)=(ucore(jp,:)-ucore(jm,:))/2; % \mu\delta
    dmu(2,j,:)=(ucore(jp,:)-2*ucore(j,:)+ucore(jm,:))/2; % \delta^2
  end% if odd/even
```

Recursively take δ^2 of these to form higher order centred differences (could unroll a little to cater for two in parallel).

```
  for k=3:patches.ordCC
    dmu(k,:,:)=dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
  end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts & Kevrekidis 2007, ?), using weights computed in `configPatches1()` . Here interpolate to specified order.

```
  if patches.EnsAve
    u(nSubP,j,:)=repmat(ucore(j)'*(1-patches.alt) ...
      +sum(bsxfun(@times,patches.Cwtsr,dmu)),[1,1,nVars]) ...
      -sum(u((nSubP-patches.nCore+1):(nSubP-1),:,:),1);
    u(1,j,:)=repmat(ucore(j)'*(1-patches.alt) ...
      +sum(bsxfun(@times,patches.Cwtsl,dmu)),[1,1,nVars]) ...
      -sum(u(2:patches.nCore,:,:),1);
  else
    u(nSubP,j,:)=ucore(j,:)*(1-patches.alt) ...
      + reshape(-sum(u((nSubP-patches.nCore+1):(nSubP-1),j,:),1) ...
      +sum(bsxfun(@times,patches.Cwtsr,dmu)),nPatch,nVars);
    u(1,j,:)=ucore(j,:)*(1-patches.alt) ...
      +reshape(-sum(u(2:patches.nCore,j,:),1)  ...
      +sum(bsxfun(@times,patches.Cwtsl,dmu)),nPatch,nVars);
  end;
```

Case of spectral interpolation Assumes the domain is macro-periodic. As the macroscale fields are N-periodic, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For `nPatch` patches we resolve 'wavenumbers' $|k| <$ `nPatch`$/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ for odd N, and $k = (0, 1, \ldots, k_{\max}, \pm(k_{\max} + 1), -k_{\max}, \ldots, -1)$ for even N.

```
else% spectral interpolation
```

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches1` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```
if patches.alt % transform by doubling the number of fields
  v=nan(size(u)); % currently to restore the shape of u
  u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
  altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
  iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
  r=r/2;           % ratio effectively halved
  nPatch=nPatch/2; % halve the number of patches
  nVars=nVars*2;   % double the number of fields
else % the values for standard spectral
  altShift=0;
  iV=1:nVars;
end
```

Now set wavenumbers.

```
kMax=floor((nPatch-1)/2);
ks=2*pi/nPatch*(mod((0:nPatch-1)+kMax,nPatch)-kMax);
```

Test for reality of the field values, and define a function accordingly.

```
if imag(u(i0,:,:))==0, uclean=@(u) real(u);
  else                 uclean=@(u) u;
  end
```

Compute the Fourier transform of the patch centre-values for all the fields. If there are an even number of points, then zero the zig-zag mode in the FT and add it in later as cosine.

```
Ck=fft(u(i0,:,:));
if mod(nPatch,2)==0
  Czz=Ck(1,nPatch/2+1,:)/nPatch;
  Ck(1,nPatch/2+1,:)=0;
end
```

The inverse Fourier transform gives the edge values via a shift a fraction r to the next macroscale grid point. Enforce reality when appropriate.

```
u(nSubP,:,iV)=uclean(ifft(bsxfun(@times,Ck ...
    ,exp(1i*bsxfun(@times,ks,altShift+r)))));
u( 1,:,iV)=uclean(ifft(bsxfun(@times,Ck ...
    ,exp(1i*bsxfun(@times,ks,altShift-r)))));
```

For an even number of patches, add in the cosine mode.

```
if mod(nPatch,2)==0
  cosr=cos(pi*(altShift+r+(0:nPatch-1)));
  u(nSubP,:,iV)=u(nSubP,:,iV)+uclean(bsxfun(@times,Czz,cosr));
  cosr=cos(pi*(altShift-r+(0:nPatch-1)));
```

```
    u( 1,:,iV)=u( 1,:,iV)+uclean(bsxfun(@times,Czz,cosr));
  end
```

Restore staggered grid when appropriate. Is there a better way to do this??

```
if patches.alt
  nVars=nVars/2;  nPatch=2*nPatch;
  v(:,1:2:nPatch,:)=u(:,:,1:nVars);
  v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
  u=v;
end
end% if spectral
```

Fin, returning the 2/3D array of field values.

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

Subsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. Section 4.4.1 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 4.4.1 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)`] × [`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.

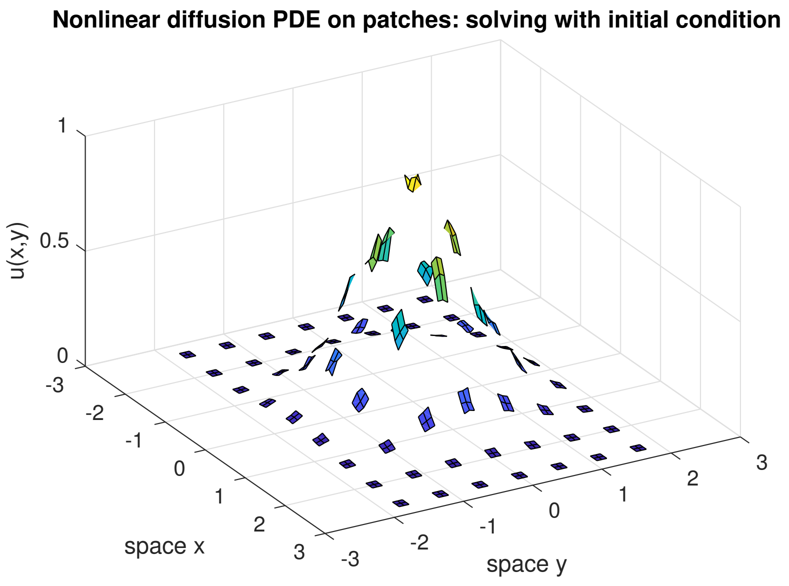
4.4.1 If no arguments, then execute an example

```
if nargin==0
```

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

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

Figure 4.2: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE: Figure 4.3 plots the computed field at time $t = 3$.



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 couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch.

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

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. Start by showing the initial conditions of Figure 4.2 while the simulation computes.

```
figure(1), clf
x=patches.x; y=patches.y;
x([1 end],:)=nan; y([1 end],:)=nan;
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)
title('Nonlinear diffusion PDE on patches: solving with initial condition')
xlabel('space x'), ylabel('space y'), zlabel('u(x,y)')
drawnow
```

Integrate in time using standard functions.

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

Figure 4.3: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear ‘diffusion’ PDE with initial condition in Figure 4.2.

```
[ts,ucts]=ode15s(@patchSmooth2,[0 3],u0(:));
```

Animate the computed simulation to end with Figure 4.3.

```
for i=1:length(ts)
  u = patchEdgeInt2(ucts(i,:));
  u = reshape(permute(u,[1 3 2 4]), [numel(x) numel(y)]);
  hsurf.ZData = u';
  title(['Nonlinear diffusion PDE on patches: time = ' num2str(ts(i))])
  pause(0.1)
end
```

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
```

4.4.2 The code to make patches

Initially duplicate parameters as needed.

```
if numel(Xlim)==2, Xlim=repmat(Xlim,1,2); end
if numel(nPatch)==1, nPatch=repmat(nPatch,1,2); end
if numel(ratio)==1, ratio=repmat(ratio,1,2); end
if numel(nSubP)==1, nSubP=repmat(nSubP,1,2); end
```

Set one edge-value to compute by interpolation if not specified by the user. Store in the struct.

```
if nargin<8, nEdge=1; end
if nEdge>1, error('multi-edge-value interp not yet implemented'), end
if 2*nEdge+1>nSubP, error('too many edge values requested'), end
patches.nEdge=nEdge;
```

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

```
patches.fun=fun;
```

Second, store the order of interpolation that is to provide the values for the inter-patch coupling conditions. Spectral coupling is `ordCC` of 0 and -1.

```
if ~ismember(ordCC,[0])
    error('ordCC out of allowed range [0]')
end
```

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

```
patches.alt=mod(ordCC,2);
ordCC=ordCC+patches.alt;
patches.ordCC=ordCC;
```

Might as well precompute the weightings for the interpolation of field values for coupling. (Could sometime extend to coupling via derivative values.)

```
ratio=ratio(:)'; % force to be row vector
if patches.alt  % eqn (7) in \cite{Cao2014a}
  patches.Cwtsr=[1
    ratio/2
    (-1+ratio.^2)/8
    (-1+ratio.^2).*ratio/48
    (9-10*ratio.^2+ratio.^4)/384
    (9-10*ratio.^2+ratio.^4).*ratio/3840
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6)/46080
    (-225+259*ratio.^2-35*ratio.^4+ratio.^6).*ratio/645120 ];
else %
  patches.Cwtsr=[ratio
    ratio.^2/2
    (-1+ratio.^2).*ratio/6
    (-1+ratio.^2).*ratio.^2/24
    (4-5*ratio.^2+ratio.^4).*ratio/120
    (4-5*ratio.^2+ratio.^4).*ratio.^2/720
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio/5040
    (-36+49*ratio.^2-14*ratio.^4+ratio.^6).*ratio.^2/40320 ];
end
patches.Cwtsr=patches.Cwtsr(1:ordCC,:);
% should avoid this next implicit auo-replication
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
```

Third, set the centre of the patches in a the macroscale grid of patches assuming periodic macroscale domain.

```
X=linspace(Xlim(1),Xlim(2),nPatch(1)+1);
X=X(1:nPatch(1))+diff(X)/2;
DX=X(2)-X(1);
Y=linspace(Xlim(3),Xlim(4),nPatch(2)+1);
Y=Y(1:nPatch(2))+diff(Y)/2;
DY=Y(2)-Y(1);
```

Construct the microscale in each patch, assuming Dirichlet patch edges, and a half-patch length of `ratio(1)` · `DX` and `ratio(2)` · `DY`.

```
nSubP=nSubP(:)'; % force to be row vector
if mod(nSubP,2)==[0 0], error('configPatches2: nSubP must be odd'), end
i0=(nSubP(1)+1)/2;
dx=ratio(1)*DX/(i0-1);
patches.x=bsxfun(@plus,dx*(-i0+1:i0-1)',X); % micro-grid
i0=(nSubP(2)+1)/2;
dy=ratio(2)*DY/(i0-1);
patches.y=bsxfun(@plus,dy*(-i0+1:i0-1)',Y); % micro-grid
end% function
```

Fin.

4.5 `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)` · `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.

Reshape the fields `u` as a 4/5D-array, and sets the edge values from macroscale interpolation of centre-patch values. Section 4.6 describes `patchEdgeInt2()`.

```
u=patchEdgeInt2(u);
```

Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero, then return to a to the user/integrator as column vector.

```
dudt=patches.fun(t,u,patches.x,patches.y);
dudt([1 end],:,:,:,:)=0;
dudt(:,[1 end],:,:,:)=0;
dudt=reshape(dudt,[],1);
```

Fin.

4.6 `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` × `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.

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

```
[ny,Ny] = size(patches.y);
[nx,Nx] = size(patches.x);
nVars = round(numel(u)/numel(patches.x)/numel(patches.y));
if numel(u) ~= nx*ny*Nx*Ny*nVars
  nSubP=[nx ny], nPatch=[Nx Ny], nVars=nVars, sizeu=size(u)
end
u = reshape(u,[nx ny Nx Ny nVars]);
```

With Dirichlet patches, the half-length of a patch is $h = dx(n_\mu - 1)/2$ (or -2 for specified flux), and the ratio needed for interpolation is then $r = h/\Delta X$. Compute lattice sizes from inside the patches as the edge values may be NaNs, etc.

```
dx = patches.x(3,1)-patches.x(2,1);
DX = patches.x(2,2)-patches.x(2,1);
rx = dx*(nx-1)/2/DX;
dy = patches.y(3,1)-patches.y(2,1);
DY = patches.y(2,2)-patches.y(2,1);
ry = dy*(ny-1)/2/DY;
```

For the moment assume the physical domain is macroscale periodic so that the coupling formulas are simplest. Should eventually cater for periodic, odd-mid-gap, even-mid-gap, even-mid-patch, dirichlet, neumann, ?? These index vectors point to patches and their two immediate neighbours.

```
%i=1:Nx; ip=mod(i,Nx)+1; im=mod(j-2,Nx)+1;
%j=1:Ny; jp=mod(j,Ny)+1; jm=mod(j-2,Ny)+1;
```

The centre of each patch (as `nx` and `ny` are odd) is at

```
i0 = round((nx+1)/2);
j0 = round((ny+1)/2);
```

Lagrange interpolation gives patch-edge values So compute centred differences of the mid-patch values for the macro-interpolation, of all fields. Assumes the domain is macro-periodic.

```
if patches.ordCC>0 % then non-spectral interpolation
error('non-spectral interpolation not yet implemented')
  dmu=nan(patches.ordCC,nPatch,nVars);
%  if patches.alt % use only odd numbered neighbours
%    dmu(1,:,:)=(u(i0,jp,:)+u(i0,jm,:))/2; % \mu
%    dmu(2,:,:)= u(i0,jp,:)-u(i0,jm,:); % \delta
%    jp=jp(jp); jm=jm(jm); % increase shifts to \pm2
%  else % standard
    dmu(1,:,:)=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
    dmu(2,:,:)=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
%  end% if odd/even
```

Recursively take δ^2 of these to form higher order centred differences (could unroll a little to cater for two in parallel).

```
  for k=3:patches.ordCC
    dmu(k,:,:)=dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
  end
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts & Kevrekidis 2007), using weights computed in `configPatches2()` . Here interpolate to specified order.

```
  u(nSubP,j,:)=u(i0,j,:)*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsr,dmu));
  u( 1,j,:)=u(i0,j,:)*(1-patches.alt) ...
    +sum(bsxfun(@times,patches.Cwtsl,dmu));
```

Case of spectral interpolation Assumes the domain is macro-periodic. We interpolate in terms of the patch index j, say, not directly in space. As the macroscale fields are N-periodic in the patch index j, the macroscale Fourier transform writes the centre-patch values as $U_j = \sum_k C_k e^{ik2\pi j/N}$. Then the edge-patch values $U_{j\pm r} = \sum_k C_k e^{ik2\pi/N(j\pm r)} = \sum_k C'_k e^{ik2\pi j/N}$ where $C'_k = C_k e^{ikr2\pi/N}$. For N patches we resolve 'wavenumbers' $|k| < N/2$, so set row vector `ks` $= k2\pi/N$ for 'wavenumbers' $k = (0, 1, \ldots, k_{\max}, -k_{\max}, \ldots, -1)$ for odd N, and $k = (0, 1, \ldots, k_{\max}, \pm(k_{\max}+1) - k_{\max}, \ldots, -1)$ for even N.

```
else% spectral interpolation
```

Deal with staggered grid by doubling the number of fields and halving the number of patches (`configPatches2` tests there are an even number of patches). Then the patch-ratio is effectively halved. The patch edges are near the middle of the gaps and swapped.

```
%  if patches.alt % transform by doubling the number of fields
%  error('staggered grid not yet implemented')
%    v=nan(size(u)); % currently to restore the shape of u
%    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
%    altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
%    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate field
%    r=r/2;           % ratio effectively halved
%    nPatch=nPatch/2; % halve the number of patches
%    nVars=nVars*2;   % double the number of fields
%  else % the values for standard spectral
    altShift = 0;
    iV = 1:nVars;
%  end
```

Now set wavenumbers in the two directions. In the case of even N these compute the $+$-case for the highest wavenumber zig-zag mode, $k = (0, 1, \ldots, k_{\max}, +(k_{\max}+1) - k_{\max}, \ldots, -1)$.

```
  kMax = floor((Nx-1)/2);
  krx = rx*2*pi/Nx*(mod((0:Nx-1)+kMax,Nx)-kMax);
  kMay = floor((Ny-1)/2);
  kry = ry*2*pi/Ny*(mod((0:Ny-1)+kMay,Ny)-kMay);
```

Test for reality of the field values, and define a function accordingly.

```
  if imag(u(i0,j0,:,:,:))==0, uclean = @(u) real(u);
     else uclean = @(u) u; end
```

Compute the Fourier transform of the patch centre-values for all the fields. If there are an even number of points, then zero the zig-zag mode in the FT and add it in later as cosine.

```
  Ck = fft2(squeeze(u(i0,j0,:,:,:)));
```

The inverse Fourier transform gives the edge values via a shift a fraction `rx/ry` to the next macroscale grid point. Initially preallocate storage for all the IFFTs that we need to cater for the zig-zag modes when there are an even number of patches in the directions.

```
nFTx = 2-mod(Nx,2);
nFTy = 2-mod(Ny,2);
unj = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
u1j = nan(1,ny,Nx,Ny,nVars,nFTx*nFTy);
uin = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);
ui1 = nan(nx,1,Nx,Ny,nVars,nFTx*nFTy);
```

Loop over the required IFFTs.

```
iFT = 0;
for iFTx = 1:nFTx
for iFTy = 1:nFTy
iFT = iFT+1;
```

First interpolate onto x-limits of the patches. (It may be more efficient to product exponentials of vectors, instead of exponential of array—only for $N > 100$. Can this be vectorised further??)

```
for jj = 1:ny
  ks = (jj-j0)*2/(ny-1)*kry; % fraction of kry along the edge
  unj(1,jj,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,altShift+krx',ks))));
  u1j(1,jj,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,altShift-krx',ks))));
end
```

Second interpolate onto y-limits of the patches.

```
for i = 1:nx
  ks = (i-i0)*2/(nx-1)*krx; % fraction of krx along the edge
  uin(i,1,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,ks',altShift+kry))));
  ui1(i,1,:,:,iV,iFT) = ifft2( bsxfun(@times,Ck ...
      ,exp(1i*bsxfun(@plus,ks',altShift-kry))));
end
```

When either direction have even number of patches then swap the zig-zag wavenumber to the conjugate.

```
if nFTy==2, kry(Ny/2+1) = -kry(Ny/2+1); end
end% iFTy-loop
if nFTx==2, krx(Nx/2+1) = -krx(Nx/2+1); end
end% iFTx-loop
```

Put edge-values into the u-array, using `mean()` to treat a zig-zag mode as cosine. Enforce reality when appropriate via `uclean()`.

```
u(end,:,:,:,iV) = uclean( mean(unj,6) );
u( 1 ,:,:,:,iV) = uclean( mean(u1j,6) );
u(:,end,:,:,iV) = uclean( mean(uin,6) );
u(:, 1 ,:,:,iV) = uclean( mean(ui1,6) );
```

Restore staggered grid when appropriate. Is there a better way to do this??

```
%if patches.alt
%  nVars=nVars/2;  nPatch=2*nPatch;
%  v(:,1:2:nPatch,:)=u(:,:,1:nVars);
%  v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
%  u=v;
%end
end% if spectral
end% function patchEdgeInt2
```

Fin, returning the 4/5D array of field values with interpolated edges.

Appendix A Create, document and test algorithms

For developers to create and document the various functions, we use an idea due to Neil D. Lawrence of the University of Sheffield:

- Each class of toolbox functions is located in separate directories in the repository, say `Dir`.
- Create a LaTeX file `Dir/funs.tex`: establish as one LaTeX section that `\input{Dir/*.m}`s the files of the functions in the class, example scripts of use, and possibly test scripts, Table A.1.
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/eqnFreeDevMan.tex` so that people can most easily work on one section at a time:
  - put `\include{funs}` into `Doc/eqnFreeDevMan.tex`;
  - to include we have to use a soft link so at the command line in the directory `Doc` execute `ln -s ../Dir/funs.tex` [1]
- Each toolbox function is documented as a separate section, with tests and examples as separate sections.
- Each function-section and test-section is to be created as a MATLAB/ Octave `Dir/*.m` file, say `Dir/fun1.m`, so that users simply invoke the function in MATLAB/Octave as usual by `fun1(...)`.

  Some editors may need to be told that `fun1.m` is a LaTeX file. For example, TexShop on the Mac requires one to execute in a Terminal

  ```
  defaults write TeXShop OtherTeXExtensions -array-add "m"
  ```
- Table A.2 gives the template for the `Dir/*.m` function-sections. The format for a example/test-section is similar.
- Any figures from examples should be generated and then saved for later inclusion with the following (which finally works properly for MATLAB 2017+)

  ```
  set(gcf,'PaperPosition',[0 0 14 10])
  print('-depsc2',filename)
  ```

  Include with `\includegraphics[scale=0.85]{filename}`

[1] Such soft links are necessary for at least my Mac OSX and hopefully will work for other developers. Further, it has the advantage that auxiliary files are also located in the `Doc` directory.

Table A.1: example `Dir/*.tex` file to typeset in the master document a function-section, say `fun.m`, and maybe the test/example-sections.

```
% input *.m files for ... Author, date
%!TEX root = ../Doc/eqnFreeDevMan.tex
\chapter{...}
\label{sec:...}
\localtableofcontents
introduction...
\input{../Dir/fun.m}
\input{../Dir/funExample.m}
...
\begin{devMan}
\section{To do}
...
\section{Miscellaneous tests}
\input{../Dir/funTest.m}
...
\end{devMan}
```

Table A.2: template for a function-section `Dir/*.m` file.

```
% Short explanation for users typing "help fun"
% Author, date
%!TEX root = ../Doc/eqnFreeDevMan.tex
%{
\section{\texttt{...}: ...}
\label{sec:...}
\localtableofcontents
Overview LaTeX explanation.
\begin{matlab}
%}
function ...
%{
\end{matlab}
\paragraph{Input} ...
\paragraph{Output} ...
\begin{devMan}
Repeated as desired:
LaTeX between end-matlab and begin-matlab
\begin{matlab}
%}
Matlab code between %} and %{
%{
\end{matlab}
Concluding LaTeX before following final lines.
\end{devMan}
%}
```

Appendix B Aspects of developing a ‘toolbox’ for patch dynamics

Chapter contents

This appendix documents sketchy further thoughts on aspects of the development.

B.1 Macroscale grid

The patches are to be distributed on a macroscale grid: the jth patch ‘centred’ at position $\vec X_j \in \mathbb{X}$. In principle the patches could move, but let’s keep them fixed in the first version. The simplest macroscale grid will be rectangular (`meshgrid`), but we plan to allow a deformed grid to secondly cater for boundary fitting to quite general domain shapes $\mathbb{X}$. And plan to later allow for more general interconnect networks for more topologies in application.

B.2 Macroscale field variables

The researcher/practitioner has to know an appropriate set of macroscale field variables $\vec U(t) \in \mathbb{R}^{d_{\vec U}}$ for each patch. For example, first they might be a simple average over a core of a patch of all of the micro-field variables; second, they might be a subset of the average micro-field variables; and third in general the macro-variables might be a nonlinear function of the micro-field variables (such as temperature is the average speed squared). The core might be just one point, or a sizeable fraction of the patch.

The mapping from microscale variable to macroscale variables is often termed the restriction.

In practice, users may not choose an appropriate set of macro-variables, so will eventually need to code some diagnostic to indicate a failure of the assumed closure.

B.3 Boundary and coupling conditions

The physical domain boundary conditions are distinct from the conditions coupling the patches together. Start with physical boundary conditions of periodicity in the macroscale.

Second, assume the physical boundary conditions are that the macro-variables are known at macroscale grid points around the boundary. Then the issue is to adjust the interpolation to cater for the boundary presence and shape. The coupling conditions for the patches should cater for the range of Robin-like boundary conditions, from Dirichlet to Neumann. Two possibilities arise: direct imposition of the coupling action (Roberts & Kevrekidis 2007), or control by the action.

Third, assume that some of the patches have some edges coincident with the boundary of the macroscale domain $\mathbb{X}$, and it is on these edges that macroscale physical boundary conditions are applied. Then the interpolation from the core of these edge patches is the same as the second case of prescribed boundary macro-variables. An issue is that each boundary patch should be big enough to cater for any spatial boundary layers transitioning from the applied boundary condition to the interior slow evolution.

Alternatively, we might have the physical boundary condition constrain the interpolation between patches.

Often microscale simulations are easiest to write when 'periodic' in microscale space. To cater for this we should also allow a control at perhaps the quartiles of a micro-periodic simulator.

B.4 Mesotime communication

Since communication limits large scale parallelism, a first step in reducing communication will be to implement only updating the coupling conditions when necessary. Error analysis indicates that updating on times longer the microscale times and shorter than the macroscale times can be effective (Bunder et al. 2016). Implementations can communicate one or more derivatives in time, as well as macroscale variables.

At this stage we can effectively parallelise over patches: first by simply using Matlab's `parfor`. Probably not using a GPU as we probably want to leave GPUs for the black box to utilise within each patch.

B.5 Projective integration

To take macroscale time steps, invoke several possible projective integration schemes: simple Euler projection, Heun-like method, etc (Samaey et al. 2010). Need to decide how long a microscale burst needs to be.

Should not need an implicit scheme as the fast dynamics are meant to be only in the micro variables, and the slow dynamics only in the macroscale variables. However, it could be that the macroscale variables have fast oscillations and it is only the amplitude of the oscillations that are slow. Perhaps need to detect and then fix or advise.

A further stage is to implement a projective integration scheme for stochastic macroscale variables: this is important because the averaging over a core of microscale roughness will almost invariably have at least some stochastic legacy effect. Calderon (2007) did some useful research on stochastic projective intergration.

B.6 Lift to many internal modes

In most problems the number of macroscale variables at any given position in space, $d_{\vec{U}}$, is less than the number of microscale variables at a position, $d_{\vec{u}}$; often much less (Kevrekidis & Samaey 2009, e.g.). In this case, every time we start a patch simulation we need to provide $d_{\vec{u}} - d_{\vec{U}}$ data at each position in the patch: this is lifting. The first methodology is to first guess, then run repeated short bursts with reinitialisation, until the simulation reaches a slow manifold. Then run the real simulation.

If the time taken to reach a local quasi-equilibrium is too long, then it is likely that the macroscale closure is bad and the macroscale variables need to be extended.

A second step is to cater for cases where the slow manifold is stochastic or is surrounded by fast waves: when it is hard to detect the slow manifold, or the slow manifold is not attractive.

B.7 Macroscale closure

In some circumstances a researcher/practitioner will not code the appropriately set of macroscale variables for a complete closure of the macroscale. For example, in thin film fluid dynamics at low Reynolds number the only macroscale variable is the fluid depth; however, at higher Reynolds number, circa ten, the inertia of the fluid becomes important and the macroscale variables must additionally include a measure of the mean lateral velocity/momentum (Roberts & Li 2006, e.g.).

At some stage we need to detect any flaw in the closure, and perhaps suggest additional appropriate macroscale variables, or at least their characteristics. Indeed, a poor closure and a stochastic slow manifold are really two faces of the same problem: the problem is that the chosen macroscale variables do not have a unique evolution in terms of themselves. A good resolution of the issue will account for both faces.

B.8 Exascale fault tolerance

Matlab is probably not an appropriate vehicle to deal with real exascale faults. However, we should cater by coding procedures for fault tolerance and testing them at least synthetically. Eventually provide hooks to a user routine to be invoked under various potential scenarios. The nature of fault tolerant algorithms will vary depending upon the scenario, even assuming that each patch burst is executed on one CPU (or closely coupled CPUs): if there are much more CPUs than patches, then maybe simply duplicate all patch simulations; if much less CPUs than patches, then an asynchronous scheduling of patch bursts should effectively cater for recomputation of failed bursts; if comparable CPUs to patches, then more subtle action is needed.

Once mesotime communication and projective integration is provided, a recomputation approach to intermittent hardware faults should be effective because we then have the tools to restart a burst from available macroscale data. Should also explore proceeding with a lower order interpolation that misses the faulty burst—because an isolated lower order interpolation probably will not affect the global order of error (it does not in approximating some boundary conditions (Gustafsson 1975, Svard & Nordstrom 2006)

B.9 Link to established packages

Several molecular/particle/agent based codes are well developed and used by a wide community of researchers. Plan to develop hooks to use some such codes as the microscale simulators on patches. First, plan to connect to LAMMPS (Plimpton et al. 2016). Second, will evaluate performance, issues, and then consider what other established packages are most promising.

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.