Equation-Free function toolbox for Matlab/Octave

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

January 21, 2019

Abstract

This 'equation-free toolbox' facilitates the computer-assisted analysis of complex, multiscale systems. Its aim is to enable 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

*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.

1 Introduction

This document is intended to be used in conjunction with the user manual. It contains line-by-line descriptions of the code in each function in the toolbox. For brief descriptions of each function, quick start guides, and many examples, see the user manual.

2 Projective integration of deterministic ODEs

Subsubsection 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.

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

Minor functions

- '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.
- Black box microsolver generator, `bbgen()`. This simple function takes as input a standard solver with a recommended time step for microscale simulation, and returns a 'black box' microsolver for the Projective Integration functions.

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

2.1 `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.

```
18 function [x, tms, xms, rm, svf] = PIRK2(solver, 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 2.1.1 as a basic template of how to use.

- `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. 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.

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(solver,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 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 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.

2.1.1 If no arguments, then execute an example

```
113 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 $\epsilon = 0.05$ using bursts of length 3ϵ.

```
epsilon = 0.05
ts = 0:6
[x,tms,xms,rm,svf] = PIRK2(@MMburst, 3*epsilon, 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
```

2.1.2 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);
```

```
195 saveFullMicro = (nArgs>3);
196 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.

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

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

```
218 tSpan(1) = tSpan(1)+bT;
219 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.

```
229 if saveMicro
230     tms = cell(nT,1);
231     xms = cell(nT,1);
232     tms{1} = reshape(relax_t,[],1);
233     xms{1} = relax_x0;
234     if saveFullMicro
235         rm.t = cell(nT,1);
236         rm.x = cell(nT,1);
237         if saveSvf
238             svf.t = nan(2*nT-2,1);
239             svf.dx = nan(2*nT-2,length(x0));
240         end
241     end
242 end
```

Loop over the macroscale time steps

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

If two 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 2*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));
        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=' nu
    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.

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

Check for round-off error.

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

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

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

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

```
334     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.

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

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

```
354         if saveFullMicro
355             rm.t{jT} = [reshape(t2,[],1); nan];
356             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
```

2.1.3 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
```

2.2 PIG(): Projective Integration with a General macrosolver

This is an approximate Projective Integration scheme in which the macrosolver is given by any user-specified scheme. Unlike the PIRK functions, PIG() cannot 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 microsolver. 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 very stiff problems, in which the burst length of the microsolver can be short, or with the 'constraint defined manifold' based microsolver provided by cdmc(), that attempts to project the variables onto the slow manifold without affecting the time.

```
14 function [t, x, tms, xms, svf] = PIG(solver,bT,macro,IC)
```

The inputs and outputs are necessarily a little different to the two PIRK2 functions.

Input

- solver(), a function that produces output from the user-specified code for micro-scale simulation. Usage:
  [tout,xout] = solver(t_in,x_in,tSim) Inputs: t_in, the initialisation time; x_in $\in \mathbb{R}^n$, the initial state; tSim, the time to simulate for. Outputs: tout, the vector of solution times, and xout, the corresponding states.

- bT, a scalar, the minimum amount of time thought needed for integration of the microsolver to relax the fast variables to the slow manifold.

The remaining inputs to PIG() set the solver and parameters used for macroscale simulation.

- macro, a struct holding information about the macrosolver.

  - macro.solver(), the numerical method that the user wants to apply on a slow time scale. The solver should be formatted as a standard numerical method in Matlab/Octave that is called as [t_out,x_out] = solver(f,tspan,IC) for an ordinary differential equation $\frac{dx}{dt} = f(t, x)$, vector of input times tspan and initial condition IC. The function f(t,x) is not an input for PIG() but will instead be estimated by PI.
  - macro.tspan, a vector of times at which the user requests output, of which the first element is always the initial time. If macro.solver can adaptively select time steps (e.g. ode45), then tspan can consist of an initial and final time only.

- IC, an n-vector of initial values at the time tspan(1).

Output Standard usage is to output only macrosolver information, with the following usage:

```
x = PIG(micro,macro,IC)
```

- x, a cell array. x{1}, an ℓ-vector of times at which PI produced output. x{2}, an $\ell \times n$ array of the approximate solution vector. Each row corresponds to an element of x{1}.

It is also possible to return the microsolver applications called by the PI method in executing the user-defined macrosolver. Much of this microscale data will not be an accurate solution of the system, but rather will consist of simulations used to relax the fast variables close to the slow manifold in the process of executing a single macroscale time step.

```
[x,xm] = PIRK4(micro,tspan,IC)
```

- xm, a cell array containing the output of all applications of the microsolver. xm{1} is an L dimensional column vector containing times; xm{2} is an $L \times n$ array of the corresponding microsolver output.

```
[x,xm,dx] = PIRK4(micro,tspan,IC)
```

- `dx`, a cell array containing the PI estimates of the slow vector field. `dx{1}` is an ℓ dimensional column vector containing all times at which the PI scheme extrapolated along microsolver data to form a macrostep. `dx{2}` is an $\ell \times n$ array containing the estimated slow vector field.

The main body of the function now follows.

Get microscale and macroscale information from inputs, and compute the number of time steps at which output is expected.

```
69 tspan = macro.tspan;
70 csolve = macro.solver;
71 nT=length(tspan)-1;
72 sIC = length(IC);
```

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

```
79 nArgs=nargout();
80 saveMicro = (nArgs>1);
81 saveSvf = (nArgs>2);
```

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

```
90 IC = reshape(IC,[],1);
91 [relax_t,relax_IC] = solver(tspan(1),IC,bT);
```

Update the initial time.

```
98 tspan(1) = tspan(1)+bT;
```

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 microsolver. Note that it is unknown a priori how many applications of the microsolver will be required; this code may be run more efficiently if the correct number is used in place of `N+1` as the dimension of the cell arrays.

```
107 if saveMicro
```

```
108     tms=cell(nT+1,1); xms=cell(nT+1,1);
109     n=1;
110     tms{n} = reshape(relax_t,[],1);
111     xms{n} = relax_IC;
112
113     if saveSvf
114         svf.t = cell(nT+1,1);
115         svf.dx = cell(nT+1,1);
116     end
117 end
```

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

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

Run a microsolver from the given initial conditions.

```
134     [t_tmp,x_micro_tmp] = solver(tt,reshape(xx,[],1),bT);
```

Compute the standard PI approximation of the slow vector field.

```
140         del = t_tmp(end)-t_tmp(end-1);
141         dx = (x_micro_tmp(end,:)-x_micro_tmp(end-1,:))'/(del);
```

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

```
147         if saveMicro
148         n=n+1;
149         tms{n} = [reshape(t_tmp,[],1); nan];
150         xms{n} = [x_micro_tmp; nan(1,sIC)];
151         if saveSvf
152             svf.t{n-1} = tt;
153             svf.dx{n-1} = dx;
154         end
155         end
```

End genProjection().

```
161 end
```

Define the approximate slow vector field according to PI.

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

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

```
177 [t,x]=feval(csolve,ff,tspan,relax_IC(end,:)');
```

Write over x(1,:) and t(1), which the user expect to be IC and tspan(1) respectively, with the given initial conditions.

```
185 x(1,:) = IC';
186 t(1) = tspan(1);
```

Output the macroscale steps:

```
193
```

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.

```
201 if saveMicro
202     tms = cell2mat(tms);
203     xms = cell2mat(xms);
204     if saveSvf
205         svf.t = cell2mat(svf.t);
206         svf.dx = cell2mat(svf.dx);
207     end
208 end
```

This concludes PIG().

```
216 end
```

2.3 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)
```

The inputs and outputs are standardised with PIRK2().

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.

2.3.1 The projective integration code

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

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

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

```
119 nArgs=nargout();
120 saveMicro = (nArgs>1);
121 saveFullMicro = (nArgs>3);
122 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.

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

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

```
144 tSpan(1) = tSpan(1)+bT;
145 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.

```
155 if saveMicro
156     tms = cell(nT,1);
157     xms = cell(nT,1);
158     tms{1} = reshape(relax_t,[],1);
159     xms{1} = relax_x0;
160     if saveFullMicro
```

```
161         rm.t = cell(nT,1);
162         rm.x = cell(nT,1);
163         if saveSvf
164             svf.t = nan(4*nT-4,1);
165             svf.dx = nan(4*nT-4,length(x0));
166         end
167     end
168 end
```

Loop over the macroscale time steps

```
176 for jT = 2:nT
177     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.

```
185     if 4*abs(bT)>=abs(tSpan(jT)-T) & bT*(tSpan(jT)-T)>0
186         [t1,xm1] = solver(T, x(jT-1,:), tSpan(jT)-T);
187         x(jT,:) = xm1(end,:);
188         t2=nan; xm2=nan(1,size(xm1,2));
189         t3=nan; t4=nan; xm3=xm2; xm4 = xm2; dx1=xm2; dx2=xm2;
190     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.

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

Check for round-off error.

```
208     xt=[reshape(t1(end-1:end),[],1) xm1(end-1:end,:)];
209     roundingTol=1e-8;
210     if norm(diff(xt))/norm(xt,'fro') < roundingTol
```

```
211     warning(['significant round-off error in 1st projection at T=' nu
212     end
```

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

```
221     Dt = tSpan(jT)-T-bT;
222     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.

```
232     xint = xm1(end,:) + (Dt/2-bT)*dx1;
233     [t2,xm2] = solver(T+Dt/2, xint, bT);
234     del = t2(end)-t2(end-1);
235     dx2 = (xm2(end,:)-xm2(end-1,:))/del;
236
237     xint = xm1(end,:) + (Dt/2-bT)*dx2;
238     [t3,xm3] = solver(T+Dt/2, xint, bT);
239     del = t3(end)-t3(end-1);
240     dx3 = (xm3(end,:)-xm3(end-1,:))/del;
241
242     xint = xm1(end,:) + (Dt-bT)*dx3;
243     [t4,xm4] = solver(T+Dt, xint, bT);
244     del = t4(end)-t4(end-1);
245     dx4 = (xm4(end,:)-xm4(end-1,:))/del;
```

Check for round-off error.

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

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

```
262     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
                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)`.

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

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

```
333 if saveMicro
334     tms = cell2mat(tms);
335     xms = cell2mat(xms);
336     if saveFullMicro
337         rm.t = cell2mat(rm.t);
338         rm.x = cell2mat(rm.x);
339     end
340 end
```

2.3.2 If no output specified, then plot simulation

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

This concludes `PIRK4()`.

```
358 end
```

2.4 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.

```
12 clear
13 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,

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

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

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

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

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

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

```
41 dt = 0.001;
42 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

```
49 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.

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

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

```
70 tSpan=0: 1 : 30;
71 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.

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

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

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

Figure 1 plots the output.

```
99  tmsr = rm.t; xmsr = rm.x;
100 clf()
101 hold on
102 PI_sol=plot(tSpan,x,'bo');
103 std_sol=plot(tt,ode45x,'r');
104 plot(tms,xms,'k.');
105 plot(tmsr,xmsr,'g.');
106 legend([PI_sol(1),std_sol(1)],'PI Solution',...
107     'Standard Solution','Location','NorthWest')
108 xlabel('Time');
109 ylabel('State');
```

Save plot to a file.

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

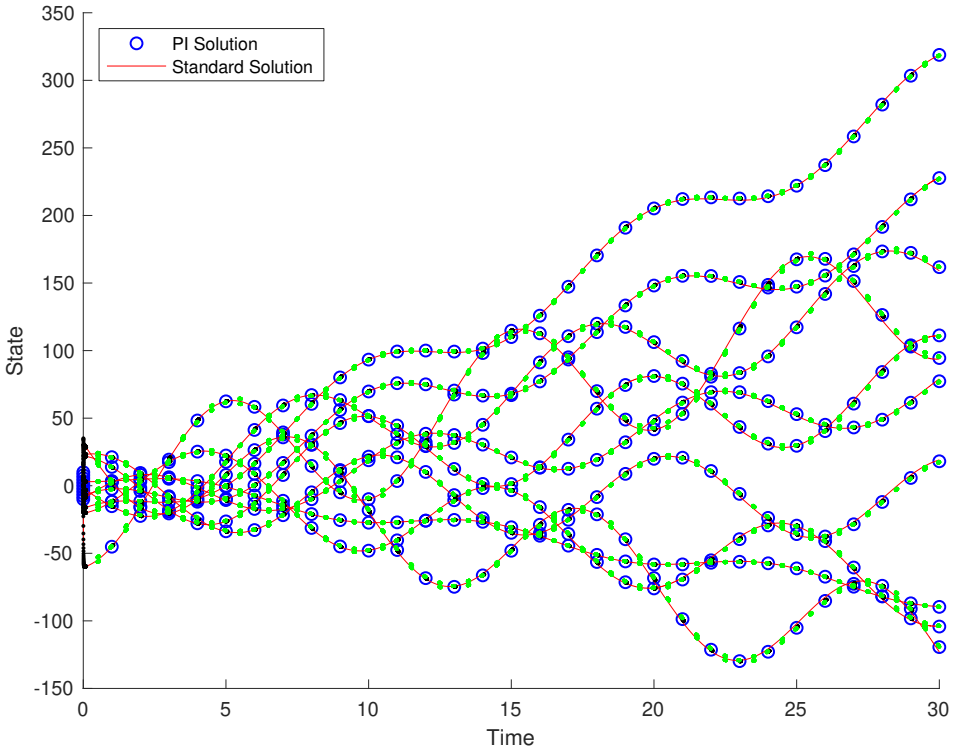


Figure 1: Demonstration of PIRK4(). From initial conditions, the system rapidly trannsitions to an attracting invariant manifold. The PI solution accurately tracks the evolution of the variables over time while requiring only a fraction of the computations of the standard solver.

2.5 Example 2: PI using General macrosolvers

In this example the PI-General scheme is applied to a singularly perturbed ordinary differential equation. The aim is to allow a standard non-stiff numerical integrator, e.g. `ode45()`, to be applied to a stiff problem on a slow, long time scale.

```
clear
```

Set time scale separation and model.

```
epsilon = 1e-4;
f=@(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.

```
solver = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
bT=20*epsilon;
```

Set initial conditions, and the time to be covered by the macrosolver. Set the macrosolver to be used as a standard, non-stiff integration scheme.

```
IC = [1 3];
tspan=[0 40];
macro.tspan = tspan;
macro.solver = 'ode45';
```

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
[t,x,tms,xms] = PIG(solver,bT,macro,IC);
tPI=toc;
fprintf(['PI took %f seconds, using ode45 as the '...
    'macrosolver.\n'],tPI)
tic
[t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
```

```
52 tODE45 = toc;
53
54 fprintf('Brute force ode45 took %f seconds.\n',tODE45)
```

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

```
63 figure; set(gcf,'PaperPosition',[0 0 14 10])
64 hold on
65 PIm=plot(tms,xms,'b.');
66 PI=plot(t,x,'bo');
67 ODE45=plot(t45,xode45,'r-','LineWidth',2);
68 legend([PI(1),ODE45(1),PIm(1)],'PI Solution',...
69     'Standard Solution','PI microsolver')
70 xlabel('Time')
71 ylabel('State')
72 axis([0 40 0 3])
73
74 figure; set(gcf,'PaperPosition',[0 0 14 10])
75 hold on
76 PI2=plot(t,x,'bo');
77 ODE452=plot(t45,xode45,'r-','LineWidth',2);
78
79 legend([PI2(1),ODE452(1)],'PI Solution','Standard Solution')
80 xlabel('Time')
81 ylabel('State')
```

The output is plotted in Figure 2.

Notes:

- the problem may be made more, or less, stiff by changing the time scale parameter `epsilon`. `PIG()` will handle a stiffer problem with ease; but if the problem is insufficiently stiff, then the algorithm will produce nonsense. This problem is handled by `cdmc()`; see Section 2.7.
- The mildly stiff problem in Example 2.4 may be efficiently solved by a standard solver, e.g. `ode45()`. The stiff but low dimensional

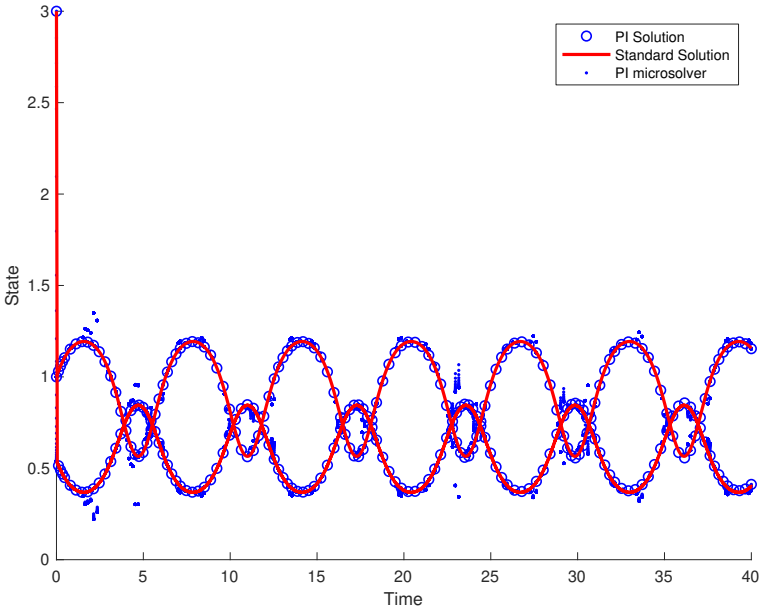


Figure 2: Accurate simulation of a stiff nonautonomous system by PIG(). The microsolver is called on-the-fly by the macrosolver (here ode45).

problem in this example can be solved efficiently by a standard stiff solver, e.g. `ode15s()`. The real advantage of the PI schemes is in high dimensional stiff problems, that cannot be efficiently solved by most standard methods.

2.6 Minor functions

2.6.1 `cdmc()`

`cdmc()` iteratively applies the microsolver 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.

Input

- `solver`, a black box microsolver suitable for PI. See any of `PIRK2(), PIRK4(), PIG()` for a description of `solver()`.
- `t`, an initial time
- `x`, an initial state
- `T`, a time period to apply `solver()` for

Output

- `tout`, a vector of times. `tout(end)` will equal `t`.
- `xout`, an array of state estimates produced by `solver()`.

This function is a wrapper for the microsolver. For instance if the problem of interest is a dynamical system that is not too stiff, and which can be simulated by the solver `sol(t,x,T)`, one would define

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

and thereafter use `csol()` in place of `sol()` as the solver for any PI scheme. The original solver `sol()` would create large errors if used in a PI scheme, but the output of `cdmc()` will not.

```
33 function [tout, xout] = cdmc(solver,t,x,T)
```

Begin with a standard application of the microsolver.

```
39     [tt,xx]=feval(solver,t,x,T);
```

Project backwards to before the initial time, then simulate one burst forwards to obtain coordinates at t.

```
49 del = (xx(end,:) - xx(end-1,:))/(tt(end) - tt(end-1));
50 b_prop = 0.2;
51 x = xx(end,:) + (1+b_prop)*(tt(1)-tt(end))*del;
```

```
52 tr=2*tt(1)-tt(end);
53 [tout,xout]=feval(solver,tr,x',b_prop*T);
```

This concludes the function.

2.6.2 bbgen()

bbgen() is a simple function that takes a standard numerical method and produces a black box solver of the type required by the PI schemes.

```
12 function bb = bbgen(solver,f,dt)
```

Input

- solver, a standard numerical solver for ordinary differential equations
- f, a function f(t,x) taking time and state inputs
- dt, a time step suitable for simulation with f

Output bb = bb(t_{in}, x_{in}, T) a 'black box' microsolver that initialises at (t_{in}, x_{in}) and simulates forward a duration T.

```
27 bb = @(t_in,x_in,T) feval(solver,f,...
28 linspace(t_in,t_in+T,1+ceil(T/dt)),x_in);
29 end
```

2.7 Explore: PI using constraint-defined manifold computing

In this example the PI-General scheme is applied to a singularly perturbed ordinary differential equation in which the time scale separation is not too strong. 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
```

Set a weak time scale separation and model.

```
epsilon = 1e-2;
f=@(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.

```
naiveSol = @(tIC, xIC,T) feval('ode45',f,[tIC tIC+T],xIC);
bT=5*epsilon;
```

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

```
cSol = @(t,x,T) cdmc(naiveSol,t,x,T);
```

Set initial conditions, and the time to be covered by the macrosolver. Set the macrosolver to be used as a standard, non-stiff integration scheme.

```
IC = [1 3];
tspan=0:0.5:40;
macro.tspan = tspan;
macro.solver = 'ode45';
```

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

```
[nt,nx] = PIG(naiveSol,bT,macro,IC);
[ct,cx] = PIG(cSol,bT,macro,IC);
[t45,xode45] = ode45(f,[tspan(1) tspan(end)],IC);
```

Plot the output.

```
figure; set(gcf,'PaperPosition',[0 0 14 10])
hold on
nPI = plot(nt,nx,'bo');
PI=plot(ct,cx,'ko');
ODE45=plot(t45,xode45,'r-','LineWidth',2);

legend([nPI(1),PI(1),ODE45(1)],'Naive PIG Solution',...
```

```
68      'PIG using cdmc','Accurate Solution')
69  xlabel('Time')
70  ylabel('State')
71  axis([0 40 0 3])
```

The output is plotted in Figure 3. 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 PI 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.

2.8 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.

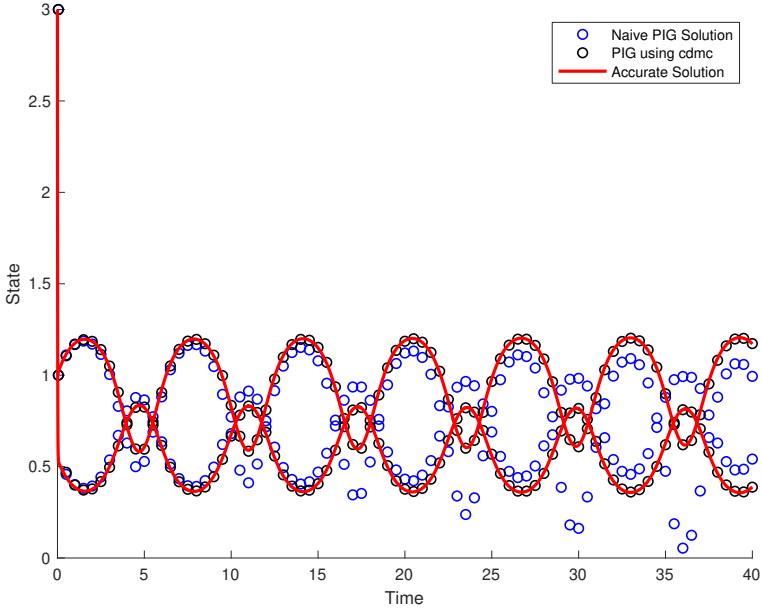


Figure 3: Accurate simulation of a weakly stiff nonautonomous system by PIG() using cdmc(), and an inaccurate solution using a naive application of PIG().

3 Patch scheme for given microscale discrete space system

Subsubsection 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 3.1.1 and 3.4.1 for basic code that uses the provided functions to simulate Burgers' PDE and a nonlinear 'diffusion' PDE.

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

Subsubsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth1()`. Section 3.1.1 lists an example of its use.

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

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

- `fun` is the name of the user function, `fun(t,u,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 in $\{-1, 0, \ldots, 8\}$.
- 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` × `nPatch` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

3.1.1 If no arguments, then execute an example

```
87 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.

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

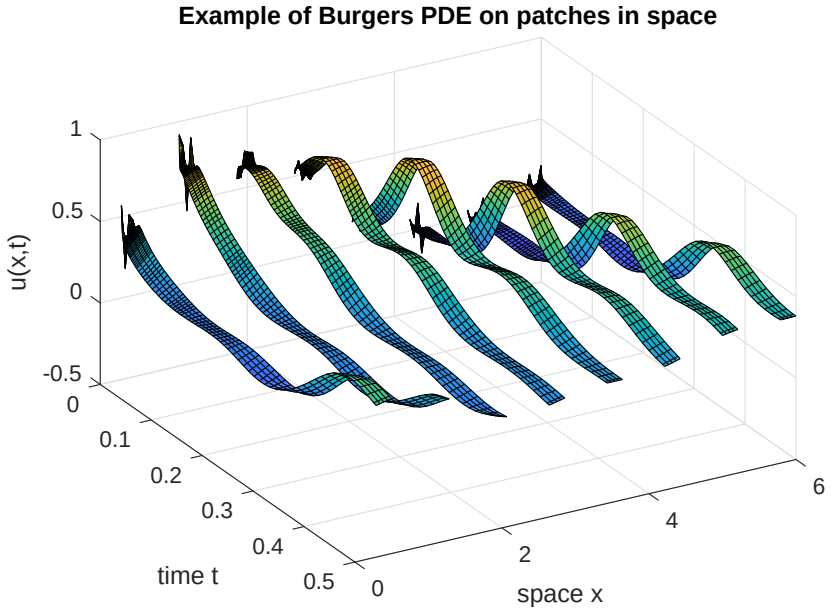
Set an initial condition, and integrate in time using standard functions.

```
113 u0=0.3*(1+sin(patches.x))+0.1*randn(size(patches.x));
114 [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 illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

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

Figure 4: field $u(x,t)$ of the patch scheme applied to Burgers' PDE.



Upon finishing execution of the example, exit this function.

```
140 return
141 end%if no arguments
```

Example of Burgers PDE inside patches As a microscale discretisation of $u_t = u_{xx} - 30uu_x$, code $\dot{u}_{ij} = \frac{1}{\delta x^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) - 30u_{ij}\frac{1}{2\delta x}(u_{i+1,j} - u_{i-1,j})$.

```
152 function ut=BurgersPDE(t,u,x)
153   dx=diff(x(1:2));  % micro-scale spacing
154   i=2:size(u,1)-1;  % interior points in patches
155   ut=nan(size(u));  % preallocate storage
156   ut(i,:)=diff(u,2)/dx^2 ...
157     -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
158 end
```

3.1.2 The code to make patches

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

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

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

```
185 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.

```
194 if ~ismember(ordCC,[-1:8])
195     error('ordCC out of allowed range [-1:8]')
196 end
```

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

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

Check for staggered grid and periodic case.

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

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

```
221 if patches.alt  % eqn (7) in \cite{Cao2014a}
222   patches.Cwtsr=[1
223     ratio/2
```

```
224     (-1+ratio^2)/8
225     (-1+ratio^2)*ratio/48
226     (9-10*ratio^2+ratio^4)/384
227     (9-10*ratio^2+ratio^4)*ratio/3840
228     (-225+259*ratio^2-35*ratio^4+ratio^6)/46080
229     (-225+259*ratio^2-35*ratio^4+ratio^6)*ratio/645120 ];
230 else %
231   patches.Cwtsr=[ratio
232     ratio^2/2
233     (-1+ratio^2)*ratio/6
234     (-1+ratio^2)*ratio^2/24
235     (4-5*ratio^2+ratio^4)*ratio/120
236     (4-5*ratio^2+ratio^4)*ratio^2/720
237     (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio/5040
238     (-36+49*ratio^2-14*ratio^4+ratio^6)*ratio^2/40320 ];
239 end
240 patches.Cwtsr=patches.Cwtsr(1:ordCC);
241 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.

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

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

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

Fin.

3.2 patchSmooth1(): interface to time integrators

Subsubsection 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.

```
23 function dudt=patchSmooth1(t,u)
24 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 micro-scales.

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 3.3 describes patchEdgeInt1().

```
68 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.

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

Fin.

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

Subsubsection contents

Couples 1D patches across 1D space by computing their edge values from macroscale interpolation. 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 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.

```
24 function u=patchEdgeInt1(u)
25 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 micro-scales.
  - `.ordCC` is order of interpolation, currently in $\{0, 2, 4, 6, 8\}$.
  - `.alt` in $\{0, 1\}$ is one for staggered grid (alternating) interpolation.
  - `.Cwtsr` and `.Cwtsl`

Output

- `u` is `nSubP` × `nPatch` × `nVars` 2/3D array of the fields with edge values set by interpolation of patch centre-values.

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

```
64 [nSubP,nPatch]=size(patches.x);
65 nVars=round(numel(u)/numel(patches.x));
66 if numel(u)~=nSubP*nPatch*nVars
67   nSubP=nSubP, nPatch=nPatch, nVars=nVars, sizeu=size(u)
68   end
69 u=reshape(u,nSubP,nPatch,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);
r=dx*(nSubP-1)/2/DX;
```

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.

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

The centre of each patch (as nSubP is odd) is at

```
i0=round((nSubP+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
  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,:);
```

```
127   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.

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

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' $|k| < \texttt{nPatch}/2$, so set row vector $\texttt{ks} = k2\pi/N$ 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.

```
155   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.

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

```
175     iV=1:nVars;
176   end
```

Now set wavenumbers.

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

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

```
190   if imag(u(i0,:,:))==0, uclean=@(u) real(u);
191     else                 uclean=@(u) u;
192     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.

```
201   Ck=fft(u(i0,:,:));
202   if mod(nPatch,2)==0
203     Czz=Ck(1,nPatch/2+1,:)/nPatch;
204     Ck(1,nPatch/2+1,:)=0;
205   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.

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

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

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

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

```
234 if patches.alt
235   nVars=nVars/2;  nPatch=2*nPatch;
236   v(:,1:2:nPatch,:)=u(:,:,1:nVars);
237   v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
238   u=v;
239 end
240 end% if spectral
```

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

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

Subsubsection contents

Makes the struct `patches` for use by the patch/gap-tooth time derivative function `patchSmooth2()`. Section 3.4.1 lists an example of its use.

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

Input If invoked with no input arguments, then executes an example of simulating a nonlinear diffusion PDE relevant to the lubrication flow of a thin layer of fluid—see Section 3.4.1 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)` × `nPatch(1)` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.
- `.y` is `nSubP(2)` × `nPatch(2)` array of the regular spatial locations y_{ij} of the microscale grid points in every patch.
- `.nEdge` is, for each patch, the number of edge values set by interpolation at the edge regions of each patch.

3.4.1 If no arguments, then execute an example

```
100 if nargin==0
```

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

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

Establish global patch data struct to interface with a function coding a nonlinear 'diffusion' PDE: to be solved on 6×4-periodic domain, with 9×7 patches, spectral interpolation couples the patches, each patch of half-size ratio 0.25, and with 5×5 points within each patch.

```
120 nSubP = 5;
121 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 5 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 condi
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')
[ts,ucts]=ode15s(@patchSmooth2,[0 3],u0(:));
```

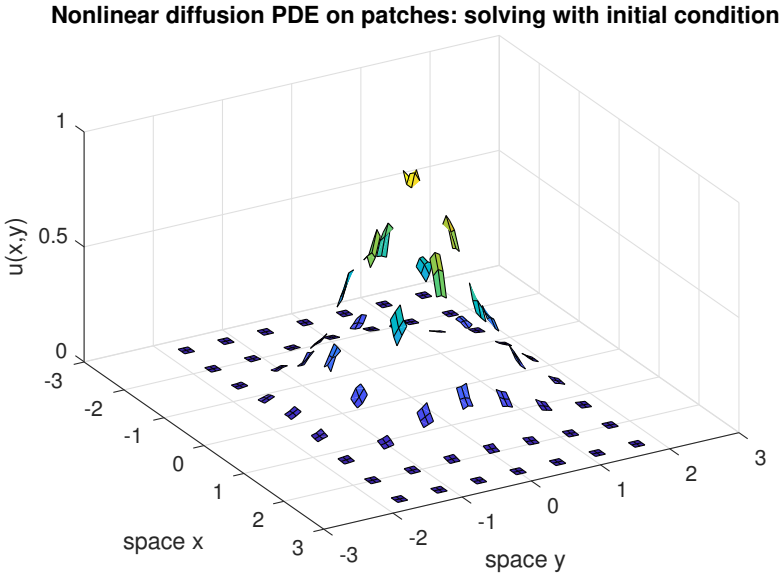
Animate the computed simulation to end with Figure 6.

```
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
```

Figure 5: initial field $u(x, y, t)$ at time $t = 0$ of the patch scheme applied to a nonlinear 'diffusion' PDE: Figure 6 plots the computed field at time $t = 3$.



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

3.4.2 The code to make patches

Initially duplicate parameters as needed.

Figure 6: field $u(x, y, t)$ at time $t = 3$ of the patch scheme applied to a nonlinear 'diffusion' PDE with initial condition in Figure 5.

```
224 if numel(Xlim)==2, Xlim=repmat(Xlim,1,2); end
225 if numel(nPatch)==1, nPatch=repmat(nPatch,1,2); end
226 if numel(ratio)==1, ratio=repmat(ratio,1,2); end
227 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.

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

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

```
247 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'), e
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.

3.5 patchSmooth2(): interface to time integrators

Subsubsection 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`.

```
23 function dudt=patchSmooth2(t,u)
24 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 micro-scales.
  - `.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 micro-scales.

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 3.6 describes patchEdgeInt2().

```
76 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.

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

Fin.

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

Subsubsection 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.

```
20 function u=patchEdgeInt2(u)
21 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 micro-scales.
  - `.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 micro-scales.
  - `.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).

```
135   for k=3:patches.ordCC
136     dmu(k,:,:)=dmu(k-2,jp,:)-2*dmu(k-2,j,:)+dmu(k-2,jm,:);
137   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.

```
145   u(nSubP,j,:)=u(i0,j,:)*(1-patches.alt) ...
146     +sum(bsxfun(@times,patches.Cwtsr,dmu));
147   u( 1,j,:)=u(i0,j,:)*(1-patches.alt) ...
148     +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.

```
169   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.

```
179   %  if patches.alt % transform by doubling the number of fields
180   %  error('staggered grid not yet implemented')
181   %    v=nan(size(u)); % currently to restore the shape of u
182   %    u=cat(3,u(:,1:2:nPatch,:),u(:,2:2:nPatch,:));
183   %    altShift=reshape(0.5*[ones(nVars,1);-ones(nVars,1)],1,1,[]);
184   %    iV=[nVars+1:2*nVars 1:nVars]; % scatter interp to alternate fiel
```

```
%    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()`.

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

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

```
300 %if patches.alt
301 %  nVars=nVars/2;  nPatch=2*nPatch;
302 %  v(:,1:2:nPatch,:)=u(:,:,1:nVars);
303 %  v(:,2:2:nPatch,:)=u(:,:,nVars+1:2*nVars);
304 %  u=v;
305 %end
306 end% if spectral
307 end% function patchEdgeInt2
```

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

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 1.
- Each such `Dir/funs.tex` file is to be included from the main LaTeX file `Doc/equationFreeDoc.tex` so that people can most easily work on one section at a time:
  - put `\include{funs}` into `Doc/equationFreeDoc.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` [2]
- Each toolbox function is documented as a separate subsection, with tests and examples as separate subsections.
- Each function-subsection and test-subsection 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 2 gives the template for the `Dir/*.m` function-subsections. The format for a example/test-subsection is similar.

[2] 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 1: example `Dir/*.tex` file to typeset in the master document a function-subsection, say `fun.m`, and maybe the test/example-subsections.

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

- 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}`

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

```
% Short explanation for users typing "help fun"
% Author, date
%!TEX root = ../Doc/equationFreeDoc.tex
%{
\subsection{\texttt{...}: ...}
\label{sec:...}
\localtableofcontents
Overview LaTeX explanation.
\begin{matlab}
%}
function ...
%{
\end{matlab}
\paragraph{Input} ...
\paragraph{Output} ...
\begin{body}
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{body}
%}
```

B Aspects of developing a 'toolbox' for patch dynamics

Section 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.

References

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.