Equation-Free function toolbox for Matlab/Octave

A. J. Roberts[*] et al.[†]

September 19, 2018

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

[*]http://www.maths.adelaide.edu.au/anthony.roberts, http://orcid.org/0000-0001-8930-1552

[†]Be the first to appear here for your contribution.

1 Introduction

Section contents

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

Blackbox scenario Assume that a researcher/practitioner has a detailed and *trustworthy* computational simulation of some problem of interest. The simulation may be written in terms of micro-positional coordinates $\vec x_i(t)$ in 'space' at which there are micro-field variable values $\vec u_i(t)$ for indices i in some (large) set of integers and for time t. In lattice problems the positions $\vec x_i$ would be fixed in time (unless employing a moving mesh on the microscale); in particle problems the positions would evolve. The positional coordinates are $\vec x_i\in\mathbb R^d$ where for spatial problems integer $d=1,2,3$, but it may be more when solving for a distribution of velocities, or pore sizes, or trader's beliefs, etc. The mirco-field variables could be in $\mathbb R^p$ for any $p=1,2,\ldots,\infty$.

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

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

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

1.1 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: **??** gives an example of the following structure to use.

- Each class of toolbox functions is located in separate directories in the repository, say `Dir`.
- Each toolbox function is documented as a separate subsection, with tests and examples as separate subsections.
- For each function, say `fun.m`, create a LaTeX file `Dir/fun.tex` of a section that `\input{Dir/*.m}`s the files of the function-subsection and the test-subsections, Table 1. Each such `Dir/fun.tex` file is to be `\include{}`ed from the main LaTeX file `equationFreeDoc.tex` so that people can most easily work on one section at a time.
- Each function-subsection and test-subsection is to be created as a Matlab/Octave `Dir/*.m` file, say `Dir/fun.m`, so that users simply invoke the function in Matlab/Octave as usual by `fun(...)`.

  Some editors may need to be told that `fun.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.
- Currently I use the beautiful `minted` package to list code, but it does require a little more effort when installing.

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

```
% input *.m files for ... Author, date
%!TEX root = ../equationFreeDoc.tex
\section{...}
\label{sec:...}
\secttoc
introduction...
\input{Dir/fun.m}
\input{Dir/funExample.m}
...
\subsection{To do}
...
```

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

```
%Short explanation for users typing "help fun"
%Author, date
%!TEX root = ../equationFreeDoc.tex
%{
\subsection{\texttt{...}: ...}
\label{sec:...}
\localtableofcontents
Summary LaTeX explanation.
\begin{matlab}
%}
function ...
%{
\end{matlab}
Repeated as desired:
LaTeX between end-matlab and begin-matlab
\begin{matlab}
%}
Matlab code between %} and %{
%{
\end{matlab}
Concluding LaTeX before following final line.
%}
```

2 Projective integration of deterministic ODEs

Section contents

This is a very first stab at a good projective integration function (Gear & Kevrekidis 2003*a*,*b*, Givon et al. 2006, e.g.).

2.1 projInt1()

This is a basic example of projective integration of a given system of stiff deterministic ODEs via DMD, the Dynamic Mode Decomposition (Kutz et al. 2016).

```
%}
function [xs,xss,tss]=projInt1(fun,x0,Ts,rank,dt,timeSteps)
%{
```

Input

- `fun()` is a function such as `dxdt=fun(t,x)` that computes the right-hand side of the ODE $d\vec x/dt=\vec f(t,\vec x)$ where $\vec x$ is a column vector, say in $\mathbb{R}^n$ for $n\geq 1$, t is a scalar, and the result $\vec f$ is a column vector in $\mathbb{R}^n$.
- `x0` is an n-vector of initial values at the time `ts(1)`. If any entries in `x0` are `NaN`, then `fun()` must cope, and only the non-`NaN` components are projected in time.
- `Ts` is a vector of times to compute the approximate solution, say in $\mathbb{R}^\ell$ for $\ell\geq 2$.

- `rank` is the rank of the DMD extrapolation over macroscale time steps. Suspect `rank` should be at least one more than the effective number of slow variables.
- `dt` is the size of the microscale time-step. Must be small enough so that RK2 integration of the ODEs is stable.
- `timeSteps` is a two element vector:
  - `timeSteps(1)` is the time thought to be needed for microscale simulation to reach the slow manifold;
  - `timeSteps(2)` is the subsequent time which DMD analyses to model the slow manifold (must be longer than `rank` $\cdot$ `dt`).

Output

- `xs`, $n \times \ell$ array of approximate solution vector at the specified times (the transpose of what MATLAB integrators do!)
- `xss`, optional, $n \times$ big array of the microscale simulation bursts—separated by NaNs for possible plotting.
- `tss`, optional, $1 \times$ big vector of times corresponding to the columns of `xss`.

Compute the time steps and create storage for outputs.

```
%}
DT=diff(Ts);
n=length(x0);
xs=nan(n,length(Ts));
xss=[];tss=[];
%{
```

If any `x0` are `NaN`, then assume the time derivative routine can cope, and here we just exclude these from DMD projection and from any error estimation. This allows a user to have space in the solutions for breaks in the data vector (that, for example, may be filled in with boundary values for a PDE discretisation).

```
%}
j=find(~isnan(x0));
%{
```

If either of the timeSteps are non-integer valued, then assume they are both times, instead of micro-time-steps, so set the number of time-steps accordingly (multiples of `dt`).

```
%}
timeSteps=round(timeSteps/dt);
timeSteps(2)=max(rank+1,timeSteps(2));
%{
```

Set an algorithmic tolerance for miscellaneous purposes. As at Jan 2018, it is a guess. It might be similar to some level of microscale 'noise' in the burst. Also, in an oscillatory mode for projection, set the expected maximum number of cycles in a projection.

```
%}
algTol=log(1e8);
cycMax=3;
%{
```

Initialise first result to the given initial condition.

```
%}
xs(:,1)=x0(:);
%{
```

Projectively integrate each of the time-steps from t_k to t_{k+1}.

```
%}
for k=1:length(DT)
%{
```

Microscale integration is simple, second order, Runge–Kutta method. Reasons: the start-up time for implicit integrators, such as ode15s, is too onerous to be worthwhile for each short burst; the microscale time step needed for stability of explicit integrators is so small that a low order method is usually accurate enough.

```
%}
x=[x0(:) nan(n,sum(timeSteps))];
for i=1:sum(timeSteps)
    xmid  =x(:,i)+dt/2*fun(Ts(k)+(i-1)*dt,x(:,i));
    x(:,i+1)=x(:,i)+dt*fun(Ts(k)+(i-0.5)*dt,xmid);
end
%{
```

If user requests microscale bursts, then store.

```
%}
if nargout>1,xss=[xss x nan(n,1)];
if nargout>2,tss=[tss Ts(k)+(0:sum(timeSteps))*dt nan];
end,end
%{
```

Grossly check on whether the microscale integration is stable. Use the 1-norm, the largest column sum of the absolute values, for little reason. Is this any use??

```
%}
if norm(x(j,ceil(end/2):end),1) ...
  > 10*norm(x(j,1:floor(end/2)),1)
  xMicroscaleIntegration=x, macroTime=Ts(k)
  warning('projInt1: microscale integration appears unstable')
  break%out of the integration loop
end
%{
```

Similarly if any non-numbers generated.

```
%}
if sum(~isfinite(x(:)))>0
  break%ou of integration loop
```

```
end
%{
```

DMD extrapolation over the macroscale But skip if the simulation has already reached the next time.

```
%}
iFin=1+sum(timeSteps);
DTgap=DT(k)-iFin*dt;
if DTgap*sign(dt)<=1e-9
   i=round(DT(k)/dt); x0(j)=x(:,i+1);
   else
%{
```

DMD appears to work better when ones are adjoined to the data vectors. [1]

```
%}
iStart=1+timeSteps(1);
x=[x;ones(1,iFin)]; j1=[j;n+1];
%{
```

Then the basic DMD algorithm: first the fit. However, need to test whether we need to worry about the microscale time step being too small and leading to an effect analogous to 'numerical differentiation' errors: akin to the rule-of-thumb in fitting chaos with time-delay coordinates that a good time-step is approximately the time of the first zero of the autocorrelation.

```
%}
[U,S,V]=svd(x(j1,iStart:iFin-1),'econ');
S=diag(S);
Sr = S(1:rank); % singular values, rx1
```

[1] A reason is as follows. Consider the one variable linear ODE $\dot x = f + J(x - x_0)$ with $x(0) = x_0$ (as from a local linearisation of nonlinear ODE). The solution is $x(t) = (x_0 - f/J) + (f/J)e^{Jt}$ which sampled at a time-step τ is $x_k = (x_0 - f/J) + (f/J)G^k$ for $G := e^{J\tau}$. Then $x_{k+1} \neq ax_k$ for any a. However, $\begin{bmatrix} x_{k+1} \\ 1 \end{bmatrix} = \begin{bmatrix} G & a \\ 0 & 1 \end{bmatrix}\begin{bmatrix} x_k \\ 1 \end{bmatrix}$ for a constant $a := (x_0 - f/J)(1 - G)$. That is, with ones adjoined, the data from the ODE fits the DMD approach.

```
AUr=bsxfun(@rdivide,x(j1,iStart+1:iFin)*V(:,1:rank),Sr.');%nxr
Atilde = U(:,1:rank)'*AUr; % low-rank dynamics, rxr
[Wr, D] = eig(Atilde); % rxr
Phi = AUr*Wr; % DMD modes, nxr
%{
```

Second, reconstruct a prediction for the time step. The current micro-simulation time is `dt*iFin`, so step forward an amount to predict the systems state at `Ts(k+1)`. Perhaps should test ω and abort if 'large' and/or positive?? Answer: not necessarily as if the rank is large then the omega could contain large negative values.

```
%}
omega = log(diag(D))/dt; % continuous-time eigenvalues, rx1
bFin=Phi\x(j1,iFin); % rx1
%{
```

But we want to neglect modes that are insignificant as characterised by Table 3, or be warned of modes that grow too rapidly. Assume appropriate to sum the neglect-ness, and the badness, for testing. Then warn if there is a mode that is too bad.

```
%}
DTgap=DT(k)-iFin*dt;
negness=max(0,-log(abs(bFin)))+max(0,-real(omega*DTgap));
badness=max(0,+real(omega*DTgap))+3/cycMax*abs(imag(omega))*DTgap;
iOK=find(negness<algTol);
iBad=find(badness(iOK)>algTol);
if ~isempty(iBad)
    warning('projInt1: some bad modes in projection')
    badness=badness(iOK(iBad))
    rank=rank
    burstDt=timeSteps*dt
    break
    end
%{
```

Table 3: criterion for deciding if some DMD modes are to be neglected, and if not neglected then are they growing too badly?

neglectness	range for $\varepsilon \approx 10^{-8}$	reason
$\max(0, -\log_e \|b_i\|)$	$0 - 19$	Very small noise in the burst implies a numerical error mode.
$\max(0, -\Re\omega_i\,\Delta t)$	$0 - 19$	Rapidly decaying mode of the macro-time-step is a micro-mode that happened to be resolved in the data.
badness		provided not already neglected
$\max(0, +\Re\omega_i\,\Delta t)$	$0 - 19$	Micro-scale mode that rapidly grows, so macro-step should be smaller.
$\frac{3}{C}\|\Im\omega_i\|\Delta t$	$0 - 19$	An oscillatory mode with $\geq C$ cycles in macro-step Δt.

Scatter the prediction into the non-Nan elements of `x0`.

```
%}
x0(j)=Phi(1:end-1,iOK)*(bFin(iOK).*exp(omega(iOK)*DTgap)); % nx1
%{
```

End the omission of the projection in the case when the burst is longer than the macroscale step.

```
%}
end
%{
```

Since some of the ω may be complex, if the simulation burst is real, then force the DMD prediction to be real.

```
%}
if isreal(x), x0=real(x0); end
xs(:,k+1)=x0;
```

```
%{
```

End the macroscale time stepping.

```
%}
end
%{
```

If requested, then add the final point to the microscale data.

```
%}
if nargout>1,xss=[xss x0];
if nargout>2,tss=[tss Ts(end)];
end,end
%{
```

End of the function with result vectors returned in columns of `xs`, one column for each time in `Ts`.

2.2 `projInt1Example1`: A first test of basic projective integration

Seek to simulate the nonlinear diffusion PDE

$$\frac{\partial u}{\partial t} = u\frac{\partial^2 u}{\partial x^2} \quad \text{such that } u(\pm 1) = 0,$$

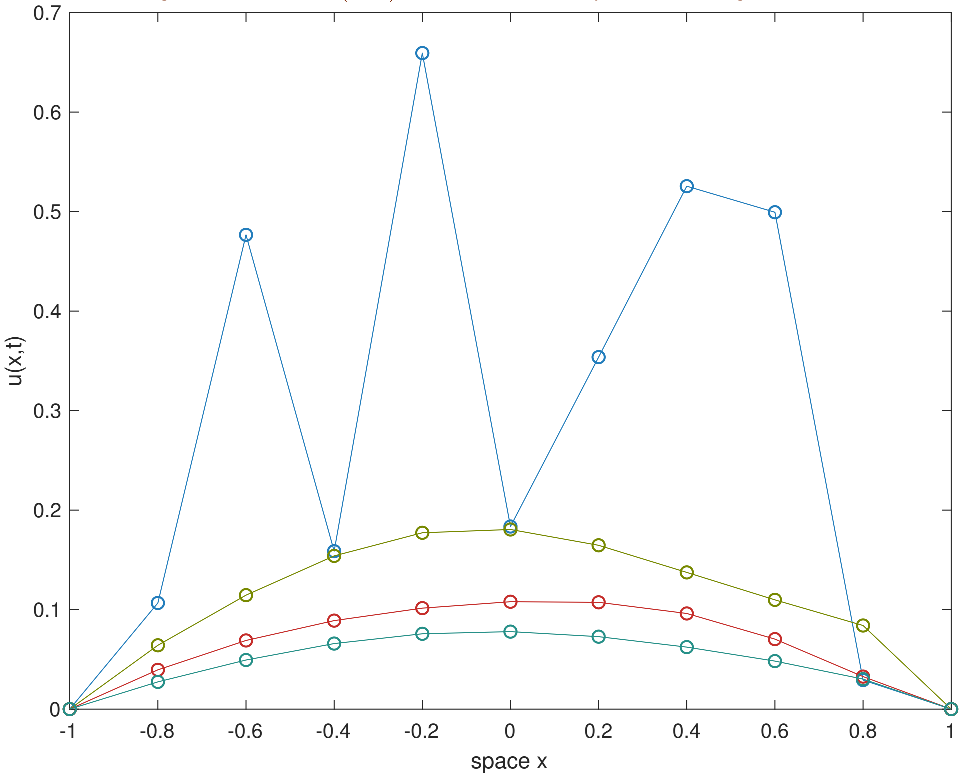
with random positive initial condition. Figure 1 shows solutions are attracted to the parabolic $u = a(t)(1 - x^2)$ with slow algebraic decay $\dot{a} = -2a^2$.

Set the number of interior points in the domain $[-1, 1]$, and the macroscale time step.

```
%}
function projInt1Example1
n=9
ts=0:2:6
%{
```

Set the initial condition to parabola or some skewed random positive values.

Figure 1: field $u(x,t)$ tests basic projective integration.



```
%}
x=linspace(-1,1,n+2)';
%u0=(1-x.^2).*(1+1e-9*randn(n+2,1));
u0=rand(n+2,1).*(1-x.^2);
%{
```

Projectively integrate in time with: rank-two DMD projection; guessed microscale time-step but chosen so an integral number of micro-steps fits into a macro-step for comparison; and guessed transient time 0.4 and 7 micro-steps ‘on the slow manifold’.

```
%}
```

```
dt=2/n^2
[us,uss,tss]=projInt1(@dudt,u0,ts,2,dt,[0.4 7*dt])
%{
```

Plot the macroscale predictions to draw Figure 1.

```
%}
clf,plot(x,us,'o-')
xlabel('space x'),ylabel('u(x,t)')
%matlab2tikz('pi1Example1u.ltx','noSize',true)
%print('-depsc2',['pi1Example1u' num2str(n)])
%{
```

Also plot a surface of the microscale bursts as shown in Figure 2.

```
%}
tss(end)=nan;% omit the last time point
clf,surf(tss,x,uss,'EdgeColor','none')
ylabel('space x'),xlabel('time t'),zlabel('u(x,t)')
view([40 30])
%print('-depsc2',['pi1Example1micro' num2str(n)])
%{
```

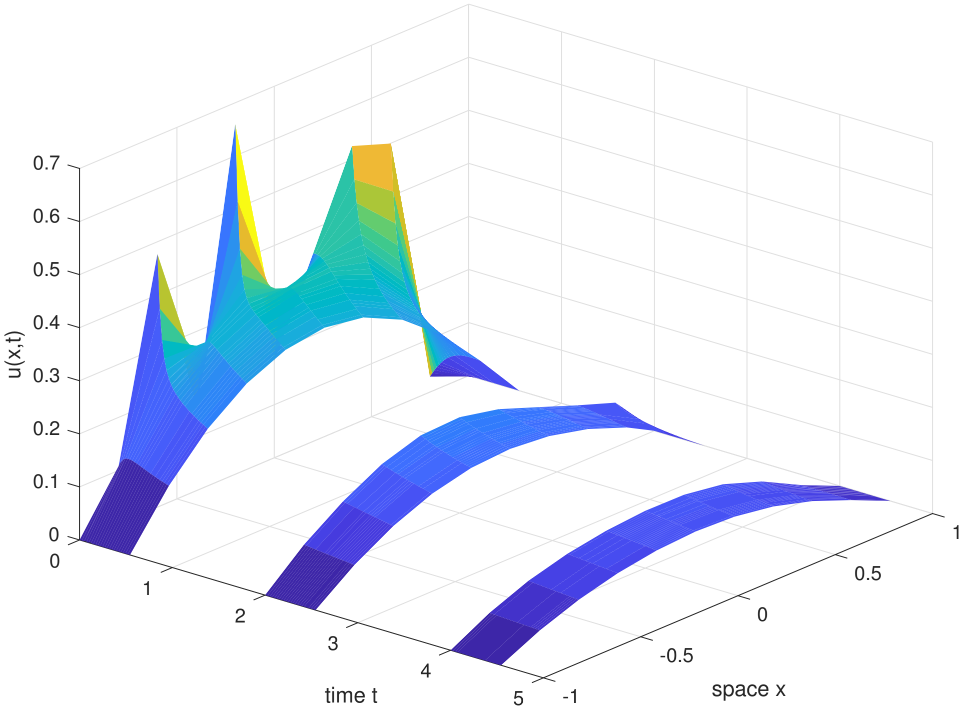
End the main function (not needed for new enough Matlab).

```
%}
end
%{
```

The nonlinear PDE discretisation Code the simple centred difference discretisation of the nonlinear diffusion PDE with constant (usually zero) boundary values.

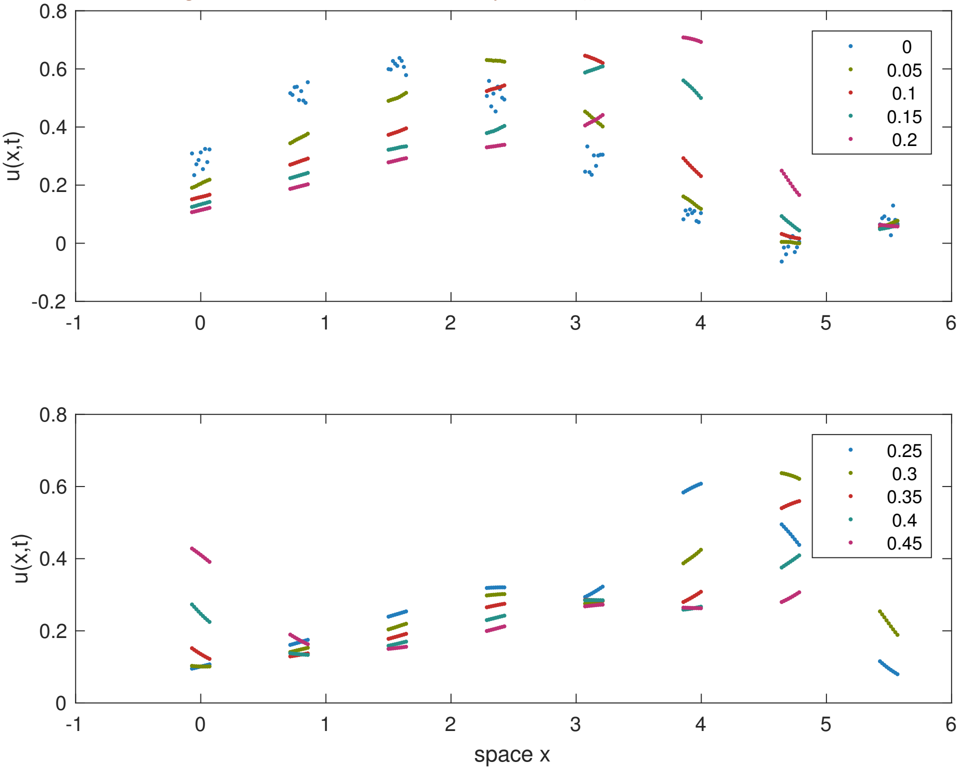
```
%}
function ut=dudt(t,u)
n=length(u);
dx=2/(n-1);
j=2:n-1;
```

Figure 2: field $u(x,t)$ during each of the microscale bursts used in the projective integration.



```
ut=[0
    u(j).*(u(j+1)-2*u(j)+u(j-1))/dx^2
    0];
end
%{
```

Figure 3: field $u(x,t)$ tests basic projective integration of a basic patch scheme of Burgers' PDE, from randomly corrupted initial conditions.



2.3 `projInt1Patches`: Projective integration of patch scheme

As an example of the use of projective integration, seek to simulate the nonlinear Burgers' PDE

$$\frac{\partial u}{\partial t} + cu\frac{\partial u}{\partial x} = \frac{\partial^2 u}{\partial x^2} \quad \text{for } 2\pi\text{-periodic } u,$$

for $c = 30$, and with various initial conditions. Use a patch scheme (Roberts & Kevrekidis 2007) to only compute on part of space as shown in Figure 3.

Function header and variables needed by discrete patch scheme.

```
%}
function projInt1Patches
global dx DX ratio j jp jm i I
%{
```

Set parameters of the patch scheme: the number of patches; number of micro-grid points within each patch; the patch size to macroscale ratio.

```
%}
nPatch=8
nSubP=11
ratio=0.1
%{
```

The points in the microscale, sub-patch, grid are indexed by `i`, and `I` is the index of the mid-patch value used for coupling patches. The macroscale patches are indexed by `j` and the neighbours by `jp` and `jm`.

```
%}
i=2:nSubP-1; % microscopic internal points for PDE
I=round((nSubP+1)/2); % midpoint of each patch
j=1:nPatch; jp=mod(j,nPatch)+1; jm=mod(j-2,nPatch)+1; % patch index
%{
```

Make the spatial grid of patches centred at X_j and of half-size $h = r\Delta X$. To suit Neumann boundary conditions on the patches make the micro-grid straddle the patch boundary by setting $dx = 2h/(n_\mu - 2)$. In order for the microscale simulation to be stable, we should have $dt \ll dx^2$. Then generate the microscale grid locations for all patches: x_{ij} is the location of the ith micro-point in the jth patch.

```
%}
X=linspace(0,2*pi,nPatch+1); X=X(j); % patch mid-points
DX=X(2)-X(1) % spacing of mid-patch points
dx=2*ratio*DX/(nSubP-2) % micro-grid size
dt=0.4*dx^2; % micro-time-step
x=bsxfun(@plus,dx*(-I+1:I-1)',X); % micro-grids
%{
```

Set the initial condition of a sine wave with random perturbations, surrounded with entries for boundary values of each patch.

```
%}
u0=[nan(1,nPatch)
    0.3*(1+sin(x(i,:)))+0.03*randn(size(x(i,:)))
    nan(1,nPatch)];
%{
```

Set the desired macroscale time-steps over the time domain.

```
%}
ts=linspace(0,0.45,10)
%{
```

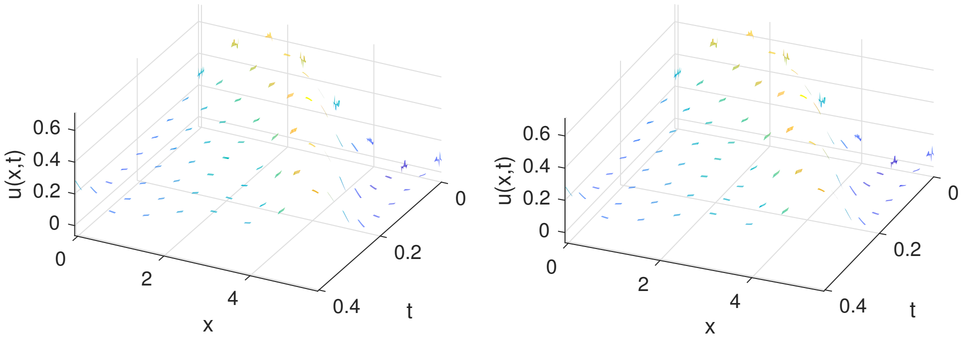
Projectively integrate in time with: DMD projection of rank nPatch + 1; guessed microscale time-step dt; and guessed numbers of transient and slow steps.

```
%}
[us,uss,tss]=projInt1(@dudt,u0(:),ts,nPatch+1,dt,[20 nPatch*2]);
%{
```

Plot the macroscale predictions to draw Figure 3, in groups of five in a plot.

```
%}
figure(1),clf
k=length(ts); ls=nan(5,ceil(k/5)); ls(1:k)=1:k;
for k=1:size(ls,2)
  subplot(size(ls,2),1,k)
  plot(x(:),us(:,ls(:,k)),'.')
  ylabel('u(x,t)')
  legend(num2str(ts(ls(:,k))'))
end
xlabel('space x')
%matlab2tikz('pi1Test1u.ltx','noSize',true)
%print('-depsc2','pi1PatchesU')
%{
```

Figure 4: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.



Also plot a surface of the microscale bursts as shown in Figure 4.

```
%}
tss(end)=nan; %omit end time-point
figure(2),clf
for k=1:2, subplot(2,2,k)
  surf(tss,x(:),uss,'EdgeColor','none')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(121-4*k,45)
end
%print('-depsc2','pi1PatchesMicro')
%{
```

End the main function (not needed for new enough Matlab).

```
%}
end
%{
```

Discretisation of Burgers PDE in coupled patches Code the simple centred difference discretisation of the nonlinear Burgers' PDE, 2π-periodic in space.

```
%}
function ut=dudt(t,u)
global dx DX ratio j jp jm i I
nPatch=j(end);
u=reshape(u,[],nPatch);
%{
```

Compute differences of the mid-patch values.

```
%}
dmu=(u(I,jp)-u(I,jm))/2; % \mu\delta
ddu=(u(I,jp)-2*u(I,j)+u(I,jm)); % \delta^2
dddmu=dmu(jp)-2*dmu(j)+dmu(jm);
ddddu=ddu(jp)-2*ddu(j)+ddu(jm);
%{
```

Use these differences to interpolate fluxes on the patch boundaries and hence set the edge values on the patch (Roberts & Kevrekidis 2007).

```
%}
u(end,j)=u(end-1,j)+(dx/DX)*(dmu+ratio*ddu ...
        -(dddmu*(1/6-ratio^2/2)+ddddu*ratio*(1/12-ratio^2/6)));
u(1,j)=u(2,j)      -(dx/DX)*(dmu-ratio*ddu ...
        -(dddmu*(1/6-ratio^2/2)-ddddu*ratio*(1/12-ratio^2/6)));
%{
```

Code Burgers' PDE in the interior of every patch.

```
%}
ut=(u(i+1,j)-2*u(i,j)+u(i-1,j))/dx^2 ...
   -30*u(i,j).*(u(i+1,j)-u(i-1,j))/(2*dx);
ut=reshape([nan(1,nPatch);ut;nan(1,nPatch)] ,[],1);
end
%{
```

2.4 `projInt1Explore1`: explore effect of varying parameters

Seek to simulate the nonlinear diffusion PDE

$$\frac{\partial u}{\partial t} = u\frac{\partial^2 u}{\partial x^2} \quad \text{such that } u(\pm 1) = 0,$$

with random positive initial condition.

Set the number of interior points in the domain $[-1, 1]$, and the macroscale time step.

```
%}
function projInt1Explore1
n=9
ts=0:2:6
dt=2/n^2
ICNoise=0.3
%{
```

Very strangely, the results from Matlab and Octave are different for the zero noise case!???? It should be deterministic. Significantly different in that Matlab fails more often.

Figure 5 shows the parameter variations when the system is already on the slow manifold. The picture after two time steps, bottom row, appears clearer than for one time step. The errors do not vary with rank provided that it is ≥ 2. There is only a very weak dependence upon the length of the burst being analysed—and that could be due to reduction in the gap. There is a weak dependence upon the transient-time, but only by a factor of two across the domain considered.

With the addition of a noisy initial conditions, Figure 6, the rank has an effect, and the transient-time appears to be a slightly stronger influence. I suspect this means that we need to allow the initial burst to have a longer transient time than subsequent bursts. Initial conditions may typically need a longer 'healing' time. Thus code an extra `timeStep` parameter.

Set the initial condition to parabola or some skewed random positive values.

Figure 5: errors in the projective integration of the nonlinear diffusion PDE from initial conditions that are on the slow manifold. Plotted are stereo views of isosurfaces in parameter space: the first row is after the first projective step; the second row after the second step.

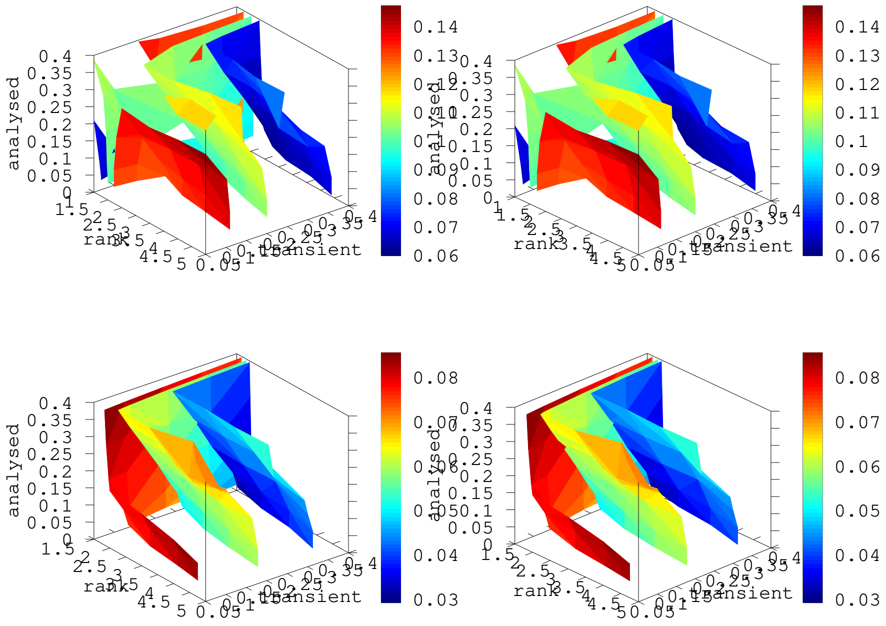
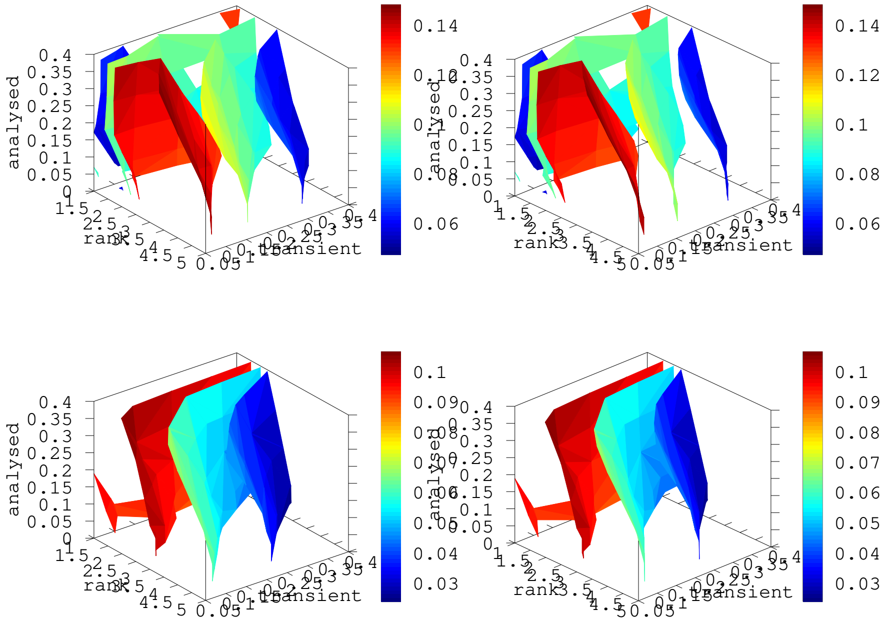


Figure 6: errors in the projective integration of the nonlinear diffusion PDE from initial conditions with noise `0.3*rand`. Plotted are stereo views of isosurfaces in parameter space: the first row is after the first projective step; the second row after the second step.



Without noise this initial condition is already on the slow manifold so only little reason for transient time.

```
%}
x=linspace(-1,1,n+2)';
u0=(0.5+ICNoise*rand(n+2,1)).*(1-x.^2);
%{
```

First find a reference solution of the microscale dynamics over all time.

```
%}
[Us,Uss,Tss]=projInt1(@dudt,u0,ts,2,dt,[0 2]);
%{
```

Set up various combinations of parameters.

```
%}
[rank,trant,slowt]=meshgrid(1:5,[1 2 4 6 8]*0.05,[2 4 8 12 16]*dt);
ps=[rank(:) trant(:) slowt(:)];
%{
```

Projectively integrate in time with various parameters.

```
%}
errs=[]; relerrs=[];
for p=ps'
[us,uss,tss]=projInt1(@dudt,u0,ts,p(1),dt,p(2:3));
%{
```

Plot the macroscale predictions

```
%}
if 0
  clf,plot(x,Us,'o-',x,us,'x--')
  xlabel('space x'),ylabel('u(x,t)')
  pause(0.01)
end
%{
```

Accumulate errors as function of time.

```
%}
err=sqrt(sum((us-Us).^2))
errs=[errs;err];
relerrs=[relerrs;err./sqrt(sum(Us.^2))];
%{
```

End the loop over parameters.

```
%}
end
%{
```

Stereo view of isosurfaces of errors after both one and two time steps. The three surfaces are the quartiles of the errors, coloured accordingly, but with a little extra colour from position for clarity.

```
%}
clf()
vals=nan(size(rank));
for k=1:2
  vals(:)=errs(:,k+1);
  q=prctile(vals(:),(0:4)*25)
  for j=1:2, subplot(2,2,j+2*(k-1)),hold on
    for i=2:4 % draw three quartiles
      isosurface(rank,trant,slowt,vals,q(i) ...
      ,q(i)+0.03*(rank/10-trant+slowt))
    end,hold off
    xlabel('rank'),ylabel('transient'),zlabel('analysed')
    colorbar
    set(gca,'view',[57-j*5,30])
  end%j
end%k
%{
```

Save to file

```
%}
print('-depsc2',['explore1icn' num2str(ICNoise*10)])
```

```
%{
```

End the main function (not needed for new enough Matlab).

```
%}
end
%{
```

The nonlinear PDE discretisation Code the simple centred difference discretisation of the nonlinear diffusion PDE with constant (usually zero) boundary values.

```
%}
function ut=dudt(t,u)
n=length(u);
dx=2/(n-1);
j=2:n-1;
ut=[0
    u(j).*(u(j+1)-2*u(j)+u(j-1))/dx^2
    0];
end
%{
```

2.5 `projInt1Explore2`: explore effect of varying parameters

Seek to simulate the nonlinear diffusion PDE

$$\frac{\partial u}{\partial t}=u\frac{\partial^2 u}{\partial x^2}\quad\text{such that } u(\pm 1)=0,$$

with random positive initial condition.

Set the number of interior points in the domain $[-1,1]$, and the macroscale time step.

```
%}
function projInt1Explore2
```

```
n=9
dt=2/n^2
ICNoise=0
%{
```

Set micro-simulation parameters. Rank two is fine when starting on the slow manifold. Choose middle of the road transient and analysed time.

```
%}
rank=2
timeSteps=[0.2 0.2]
%{
```

Try integrating with macro time-steps up to this sort of magnitude.

```
%}
Ttot=9
%{
```

Set the initial condition to parabola or some skewed random positive values. Without noise this initial condition is already on the slow manifold so only little reason for transient time.

```
%}
x=linspace(-1,1,n+2)';
u0=(0.5+ICNoise*rand(n+2,1)).*(1-x.^2);
%{
```

First find a reference solution of the microscale dynamics over all time, here stored in `Uss`.

```
%}
[Us,Uss,Tss]=projInt1(@dudt,u0,[0 Ttot],2,dt,[0 Ttot]);
%{
```

Projectively integrate two steps in time with various parameters. But remember that `projInt1` rounds `timeSteps` etc to nearest multiple of `dt`, so some of the following is a little dodgy but should not matter for overall trend.

```
%}
Dts=0.1*[1 2 4 6 10 16 26]
errs=[]; relerrs=[]; DTs=[];
for p=Dts
[~,j]=min(abs(sum(timeSteps)+p-Tss))
ts=Tss(j)*(0:2)
js=1+(j-1)*(0:2);
[us,uss,tss]=projInt1(@dudt,u0,ts,rank,dt,timeSteps);
%{
```

Plot the macroscale predictions

```
%}
if 1
  clf,plot(x,Uss(:,js),'o-',x,us,'x--')
  xlabel('space x'),ylabel('u(x,t)')
  pause(0.01)
end
%{
```

Accumulate errors as function of time.

```
%}
err=sqrt(sum((us-Uss(:,js)).^2))
errs=[errs;err];
relerrs=[relerrs;err./sqrt(sum(Uss(:,js).^2))];
%{
```

End the loop over parameters.

```
%}
end
%{
```

Plot errors

```
%}
loglog(Dts,errs(:,2:3),'o:')
xlabel('projective time-step')
```

```
ylabel('steps error')
legend('one','two')
grid
matlab2tikz('pi1x2.ltx')
%{
```

End the main function (not needed for new enough Matlab).

```
%}
end
%{
```

The nonlinear PDE discretisation Code the simple centred difference discretisation of the nonlinear diffusion PDE with constant (usually zero) boundary values.

```
%}
function ut=dudt(t,u)
n=length(u);
dx=2/(n-1);
j=2:n-1;
ut=[0
    u(j).*(u(j+1)-2*u(j)+u(j-1))/dx^2
    0];
end
%{
```

2.6 To do

- Check the order of accuracy of the algorithm.
- Develop theory quantitively justifying the DMD approach.
- Develop techniques to automatically make some of the decisions about step-sizes, burst lengths, rank, and so on.

- Develop higher accuracy versions (once we have some idea about current accuracy).
- Adapt approach to algorithms for stochastic systems.

3 Patch scheme for given microscale discrete space system

Section 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 discrete system is to be on a lattice such as obtained from finite difference approximation of a PDE. Usually continuous in time.

3.1 `patchSmooth1()`

Couples patches across space so a spatially discrete system can be integrated in time via the patch or gap-tooth scheme (Roberts & Kevrekidis 2007). Need to pass patch-design variables to this function, so use the global struct `patches`.

```
%}
function dudt=patchSmooth1(t,u)
```

```
global patches
%{
```

Input

- `u` is a vector of length `nSubP` · `nPatch` · `nVars` where there are `nVars` field values at each of the points in the `nSubP` × `nPatch` grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `makePatches()` with the following information.
  - `.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 the patch edge values will be 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 a regular lattice on both macro- and micro-scales.
  - `.ordCC`
  - `.alt`
  - `.Cwtsr` and `.Cwtsl`

Output

- `dudt` is `nSubP` · `nPatch` · `nVars` vector of time derivatives, but with zero on patch edges??

Try to figure out sizes of things. Any error arising in the reshape indicates `u` has the wrong length.

```
%}
[nM,nP]=size(patches.x);
nV=round(numel(u)/numel(patches.x));
```

```
u=reshape(u,nM,nP,nV);
%{
```

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*(nM-1)/2/DX;
%{
```

For the moment?? assume the physical domain is macroscale periodic so that the coupling formulas are simplest. Should 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:nP; jp=mod(j,nP)+1; jm=mod(j-2,nP)+1;
%{
```

The centre of each patch, assuming odd `nM`, is at

```
%}
i0=round((nM+1)/2);
%{
```

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

```
%}
dmu=nan(patches.ordCC,nP,nV);
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
%{
```

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

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

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

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

Ask the user for the time derivatives computed in the array, overwrite its edge values, then return to an integrator as column vector.

```
%}
dudt=patches.fun(t,u,patches.x);
dudt([1 nM],:,:)=0;
dudt=reshape(dudt,[],1);
%{
```

Fin.

3.2 `makePatches()`: makes the spatial patches for the suite

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

```
%}
function makePatches(fun,Xa,Xb,nPatch,ordCC,ratio,nSubP)
global patches
%{
```

Input

- `fun` is the name of the user function, `fun(t,u,x)`, that will compute time derivatives of quantities on the patches.
- `Xa,Xb` give the macro-space domain of the computation: patches are spread evenly over the interior of the interval [`Xa`, `Xb`]. Currently the system is assumed macro-periodic in this domain.
- `nPatch` is the number of evenly 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 $\{2,4,6\}$.
- `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 microscale lattice points in each patch. Must be odd so that there is a central lattice point.

Output The *global* struct `patches` is created and set.

- `patches.fun` is the name of the user's function `fun(u,t,x)` that computes the time derivatives on the patchy lattice.
- `patches.ordCC` is the specified order of inter-patch coupling.

- `patches.alt` is true for interpolation using only odd neighbouring patches as for staggered grids, and false for the usual case of all neighbour coupling.
- `patches.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.
- `patches.x` is `nSubP` $\times$ `nPatch` array of the regular spatial locations x_{ij} of the microscale grid points in every patch.

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

```
%}
patches.fun=fun;
%{
```

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

```
%}
if ~ismember(ordCC,[1:8])
    error('makePatch: ordCC out of allowed range [1:8]')
end
%{
```

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

```
%}
patches.alt=rem(ordCC,2);
ordCC=ordCC+patches.alt;
patches.ordCC=ordCC;
%{
```

Might as well precompute the weightings for the interpolation of field values for coupling. (What about coupling via derivative values?? what about spectral coupling??)

```
%}
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);
patches.Cwtsl=(-1).^((1:ordCC)'-patches.alt).*patches.Cwtsr;
%{
```

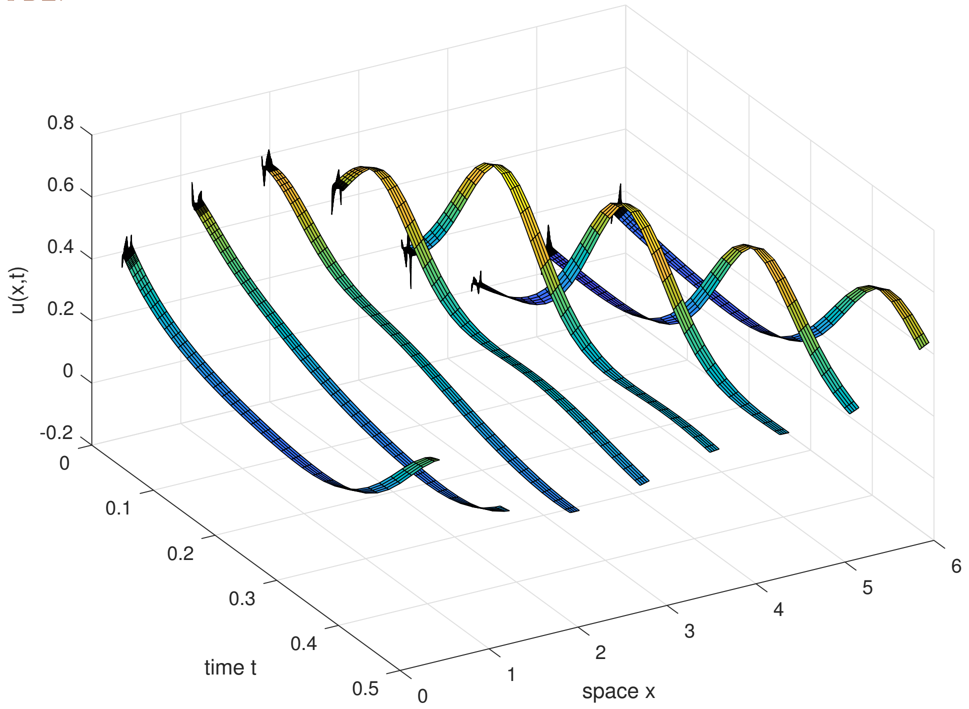
Third, set the centre of the patches in a the macroscale grid of patches assuming periodic macroscale domain.

```
%}
X=linspace(Xa,Xb,nPatch+1);
X=X(1:nPatch)+diff(X)/2;
DX=X(2)-X(1);
%{
```

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

```
%}
```

Figure 7: field $u(x,t)$ tests the patch scheme function applied to Burgers' PDE.



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

Fin.

3.3 BurgersExample: simulate Burgers' PDE on patches

Figure 7 shows an example simulation in time generated by the patch scheme function applied to Burgers' PDE. The inter-patch coupling is realised by fourth-order interpolation to the patch edges of the mid-patch values.

```
%}
function BurgersExample
%{
```

Establish global data struct for Burgers' PDE solved on 2π-periodic domain, with eight patches, each patch of half-size 0.1, with eleven points within each patch, and fourth order interpolation provides values for the inter-patch coupling conditions.

```
%}
global patches
nPatch=8
ratio=0.1
nSubP=7
Len=2*pi;
makePatches(@burgerspde,0,Len,nPatch,4,ratio,nSubP);
%{
```

Set an initial condition, and test evaluation of the time derivative.

```
%}
u0=0.3*(1+sin(patches.x))+0.05*randn(size(patches.x));
dudt=patchSmooth1(0,u0(:));
%{
```

Conventional integration in time Integrate in time using standard Matlab/Octave functions.

```
%}
ts=linspace(0,0.5,60);
if exist('OCTAVE_VERSION', 'builtin') % Octave version
   ucts=lsode(@(u,t) patchSmooth1(t,u),u0(:),ts);
else % Matlab version
   [ts,ucts]=ode15s(@patchSmooth1,ts([1 end]),u0(:));
end
```

```
%{
```

Plot the simulation.

```
%}
figure(1),clf
xs=patches.x; xs([1 end],:)=nan;
surf(ts,xs(:),ucts')
xlabel('time t'),ylabel('space x'),zlabel('u(x,t)')
view(60,40)
%print('-depsc2','ps1BurgersCtsU')
%{
```

Use projective integration Now wrap around the patch time derivative function, patchSmooth1, the projective integration function for patch simulations as illustrated by Figure 8.

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

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

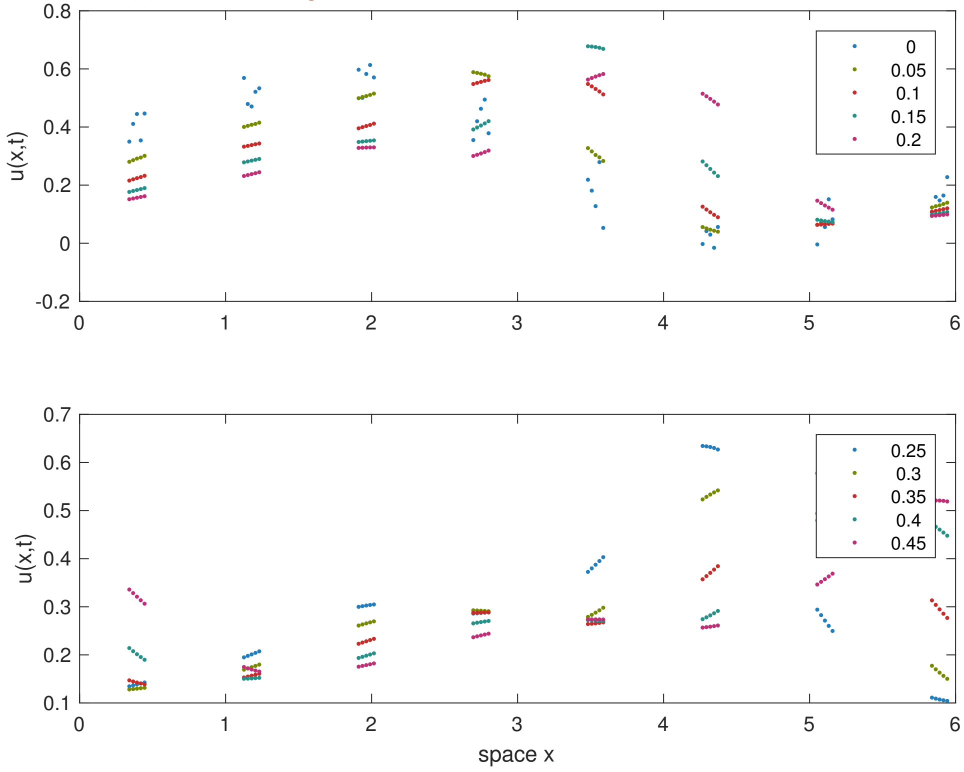
Set the desired macro- and micro-scale time-steps over the time domain.

```
%}
ts=linspace(0,0.45,10)
dt=0.4*(ratio*Len/nPatch/(nSubP/2-1))^2;
%{
```

Projectively integrate in time with: DMD projection of rank nPatch + 1; guessed microscale time-step dt; and guessed numbers of transient and slow steps.

```
%}
addpath('../ProjInt')
[us,uss,tss]=projInt1(@patchSmooth1,u0(:),ts ...
```

Figure 8: field $u(x,t)$ tests basic projective integration of the patch scheme function applied to Burgers' PDE.

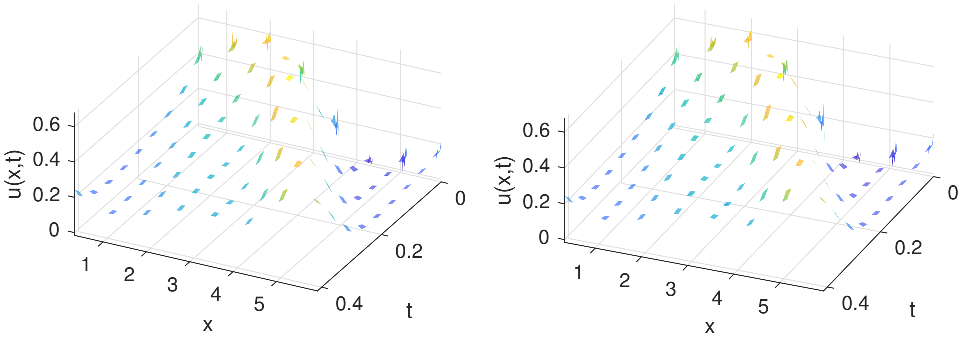


```
    ,nPatch+1,dt,[20 nPatch*2]);
%{
```

Plot the macroscale predictions to draw Figure 8, in groups of five in a plot.

```
%}
figure(2),clf
k=length(ts); ls=nan(5,ceil(k/5)); ls(1:k)=1:k;
for k=1:size(ls,2)
  subplot(size(ls,2),1,k)
  plot(xs(:),us(:,ls(:,k)),'.')
```

Figure 9: stereo pair of the field $u(x, t)$ during each of the microscale bursts used in the projective integration.



```
  ylabel('u(x,t)')
  legend(num2str(ts(ls(:,k))'))
end
xlabel('space x')
%print('-depsc2','ps1BurgersU')
%{
```

Also plot a surface of the microscale bursts as shown in Figure 9.

```
%}
tss(end)=nan; %omit end time-point
figure(3),clf
for k=1:2, subplot(2,2,k)
  surf(tss,xs(:),uss,'EdgeColor','none')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(121-4*k,45)
end
%print('-depsc2','ps1BurgersMicro')
%{
```

End the main function

```
%}
end
```

```
%{
```

This function codes the lattice equation inside the patches.

```
%}
function ut=burgerspde(t,u,x)
dx=x(2)-x(1);
ut=nan(size(u));
i=2:size(u,1)-1;
ut(i,:)=diff(u,2)/dx^2 ...
   -30*u(i,:).*(u(i+1,:)-u(i-1,:))/(2*dx);
end
%{
```

Fin.

3.4 HomogenisationExample: simulate heterogeneous diffusion in 1D on patches

Figure 10 shows an example simulation in time generated by the patch scheme function applied to heterogeneous diffusion. The inter-patch coupling is realised by fourth-order interpolation to the patch edges of the mid-patch values.

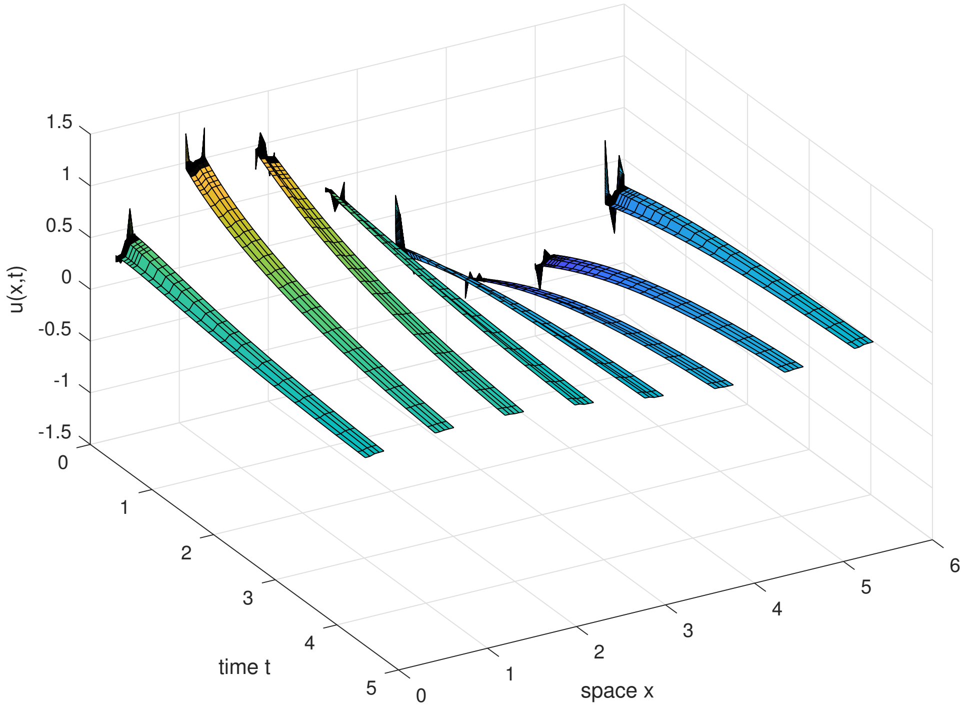
```
%}
function HomogenisationExample
%{
```

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

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

The macroscale, homogenised, effective diffusion should be the harmonic mean of these coefficients. Set the desired microscale periodicity, and microscale diffusion coefficients (with subscripts shifted by a half).

Figure 10: field $u(x,t)$ tests the patch scheme function applied to heterogeneous diffusion.



```
%}
mPeriod=3
cDiff=2*rand(mPeriod,1)
cHomo=1/mean(1./cDiff)
%{
```

Establish global data struct for heterogeneous diffusion solved on 2π-periodic domain, with eight patches, each patch of half-size 0.1, and the number of points in a patch being one more than an even multiple of the microscale periodicity (which Bunder et al. (2017) showed is accurate). Fourth order interpolation provides values for the inter-patch coupling conditions.

```
%}
```

```
global patches
nPatch=8
ratio=0.2
nSubP=2*mPeriod+1
Len=2*pi;
makePatches(@heteroDiff,0,Len,nPatch,4,ratio,nSubP);
%{
```

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

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

Set an initial condition, and test evaluation of the time derivative.

```
%}
u0=sin(patches.x)+0.2*randn(size(patches.x));
dudt=patchSmooth1(0,u0(:));
%{
```

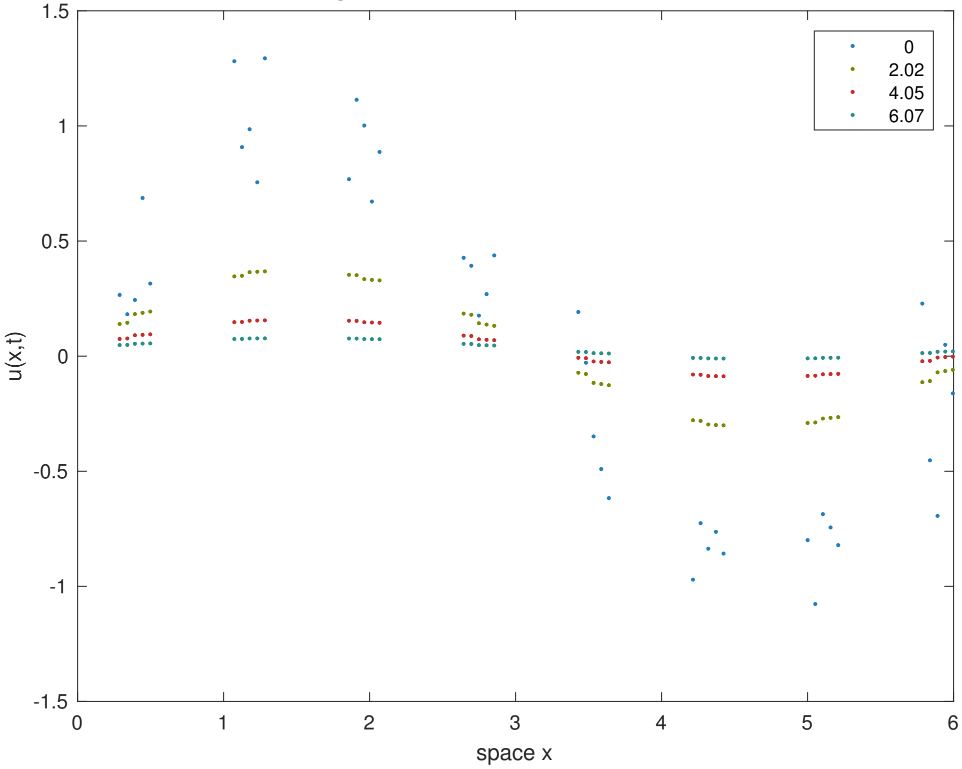
Conventional integration in time Integrate in time using standard Matlab/Octave functions.

```
%}
ts=linspace(0,2/cHomo,60);
if exist('OCTAVE_VERSION', 'builtin') % Octave version
   ucts=lsode(@(u,t) patchSmooth1(t,u),u0(:),ts);
else % Matlab version
   [ts,ucts]=ode15s(@patchSmooth1,ts([1 end]),u0(:));
end
%{
```

Plot the simulation.

```
%}
```

Figure 11: field $u(x,t)$ tests basic projective integration of the patch scheme function applied to heterogeneous diffusion.



```
figure(1),clf
xs=patches.x; xs([1 end],:)=nan;
surf(ts,xs(:),ucts')
xlabel('time t'),ylabel('space x'),zlabel('u(x,t)')
view(60,40)
%print('-depsc2','ps1HomogenisationCtsU')
%{
```

Use projective integration Now wrap around the patch time derivative function, `patchSmooth1`, the projective integration function for patch simulations as illustrated by Figure 11.

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

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

Set the desired macro- and micro-scale time-steps over the time domain.

```
%}
ts=linspace(0,3/cHomo,4)
dt=0.4*(ratio*Len/nPatch/(nSubP/2-1))^2/max(cDiff);
%{
```

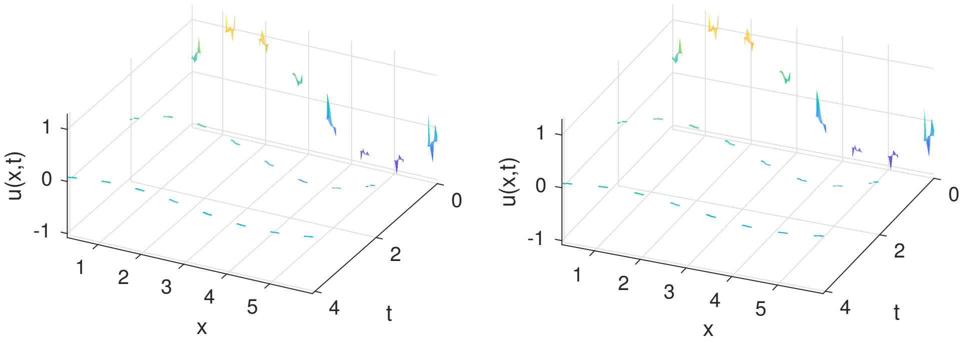
Projectively integrate in time with: DMD projection of rank `nPatch` $+ 1$; guessed microscale time-step `dt`; and guessed numbers of transient and slow steps.

```
%}
addpath('../ProjInt')
[us,uss,tss]=projInt1(@patchSmooth1,u0(:),ts ...
    ,nPatch+1,dt,[20 nPatch*2]);
%{
```

Plot the macroscale predictions to draw Figure 11, in groups of five in a plot.

```
%}
figure(2),clf
k=length(ts); ls=nan(5,ceil(k/5)); ls(1:k)=1:k;
for k=1:size(ls,2)
  subplot(size(ls,2),1,k)
  plot(xs(:),us(:,ls(~isnan(ls(:,k)),k)),'.')
  ylabel('u(x,t)')
  legend(num2str(ts(ls(~isnan(ls(:,k)),k))',3))
end
```

Figure 12: stereo pair of the field $u(x, t)$ during each of the microscale bursts used in the projective integration.



```
xlabel('space x')
%print('-depsc2','ps1HomogenisationU')
%{
```

Also plot a surface of the microscale bursts as shown in Figure 12.

```
%}
tss(end)=nan; %omit end time-point
figure(3),clf
for k=1:2, subplot(2,2,k)
  surf(tss,xs(:),uss,'EdgeColor','none')
  ylabel('x'),xlabel('t'),zlabel('u(x,t)')
  axis tight, view(121-4*k,45)
end
%print('-depsc2','ps1HomogenisationMicro')
%{
```

End the main function

```
%}
end
%{
```

This function codes the lattice heterogeneous diffusion inside the patches.

```
%}
function ut=heteroDiff(t,u,x)
global patches
dx=patches.x(2)-patches.x(1);
ut=nan(size(u));
i=2:size(u,1)-1;
ut(i,:)=diff(bsxfun(@times,patches.c,diff(u)))/dx^2 ;
end
%{
```

Fin.

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

Subsection contents

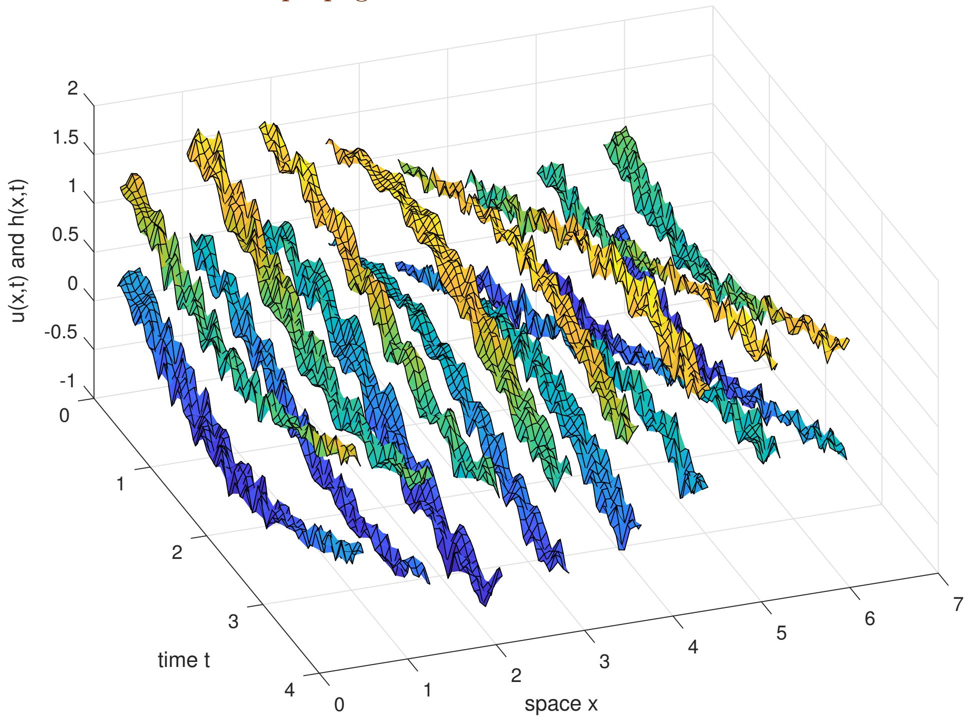
Figure 13 shows an example simulation in time generated by the patch scheme function applied to a simple wave PDE. The inter-patch coupling is realised by third-order interpolation to the patch edges of the mid-patch values.

This section describes the nonlinear microscale simulator of the nonlinear shallow water wave PDE derived from the Smagorinski model of turbulent flow (Cao & Roberts 2012, 2016*a*). Often, wave-like systems are written in terms of two conjugate variables, for example, position and momentum density, electric and magnetic fields, and water depth $h(x,t)$ and mean lateral velocity $u(x,t)$ as herein. The approach applies to any wave-like system in the form

$$\frac{\partial h}{\partial t} = -c_1\frac{\partial u}{\partial x} + f_1[h,u] \quad\text{and}\quad \frac{\partial u}{\partial t} = -c_2\frac{\partial h}{\partial x} + f_2[h,u], \tag{1}$$

where the brackets indicate that the nonlinear functions f_ℓ may involve various spatial derivatives of the fields $h(x,t)$ and $u(x,t)$. Specifically, this

Figure 13: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme function applied to simple wave PDE. A random component to the initial condition has long lasting effects on the simulation—but the macroscale wave still propagates.



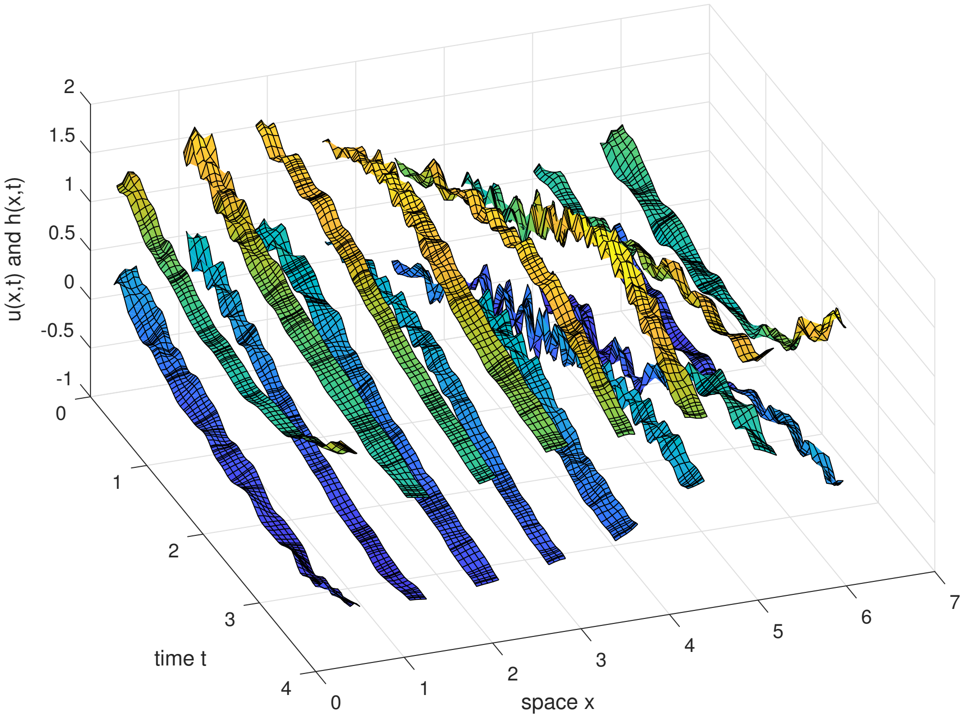
section invokes a nonlinear Smagorinski model of turbulent shallow water (Cao & Roberts 2012, 2016*a*, e.g.) along an inclined flat bed: let x measure position along the bed and in terms of fluid depth $h(x,t)$ and depth-averaged lateral velocity $u(x,t)$ the model PDEs are

$$\frac{\partial h}{\partial t} = -\frac{\partial (hu)}{\partial x}\,, \tag{2a}$$

$$\frac{\partial u}{\partial t} = 0.985\left(\tan\theta - \frac{\partial h}{\partial x}\right) - 0.003\frac{u|u|}{h} - 1.045u\frac{\partial u}{\partial x} + 0.26h|u|\frac{\partial^2 u}{\partial x^2}\,, \tag{2b}$$

where $\tan\theta$ is the slope of the bed. Equation (2a) represents conserva-

Figure 14: water depth $h(x,t)$ (above) and velocity field $u(x,t)$ (below) of the gap-tooth scheme the shallow water wave PDEs (2). A random component decays where the speed is non-zero.



tion of the fluid. The momentum PDE (2b) represents the effects of turbulent bed drag $u|u|/h$, self-advection $u\partial u/\partial x$, nonlinear turbulent dispersion $h|u|\partial^2 u/\partial x^2$, and gravitational hydrostatic forcing $\tan\theta - \partial h/\partial x$. Figure 14 shows one simulation of this system—for the same initial condition as Figure 13.

For such wave systems, let's try a staggered microscale grid and staggered macroscale patches as introduced in Figures 3 and 4, respectively, by Cao & Roberts (2016*b*).

```
%}
```

```
function waterWaveExample
%{
```

Establish global data struct for the PDEs (2) solved on 2π-periodic domain, with eight patches, each patch of half-size 0.2, with eleven points within each patch, and third-order interpolation provides values for the inter-patch coupling conditions (higher order interpolation is smoother for these smooth initial conditions).

```
%}
global patches
nPatch=8
ratio=0.2
nSubP=11 % of form 4*?-1
Len=2*pi;
makePatches(@simpleWavepde,0,Len,nPatch,3,ratio,nSubP);
%{
```

Identify which microscale grid points are h or u values. Also store them in the struct `patches` for use by the time derivative function.

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

Set an initial condition of a progressive wave, and check evaluation of the time derivative. The capital letter `U` denotes an array of values merged from both u and h fields on the staggered grids.

```
%}
U0=nan(nSubP,nPatch);
U0(hPts)=1+0.5*sin(patches.x(hPts));
U0(uPts)=0+0.5*sin(patches.x(uPts));
U0=U0+0.05*randn(nSubP,nPatch);
dUdt0=patchSmooth1(0,U0(:));% check
```

```
%dUdt0=reshape(dUdt0,nSubP,nPatch)
%{
```

Conventional integration in time Integrate in time using standard Matlab/Octave functions the two cases of the simple wave equations and the water wave equations.

```
%}
for k=1:2
%{
```

When using `ode15s` we subsample the results because the sub-grid scale waves do not dissipate and so the integrator takes very small time steps for all time.

```
%}
ts=linspace(0,4,41);
if exist('OCTAVE_VERSION', 'builtin') % Octave version
   Ucts=lsode(@(u,t) patchSmooth1(t,u),U0(:),ts);
else % Matlab version
    [ts,Ucts]=ode15s(@patchSmooth1,ts([1 end]),U0(:));
   ts=ts(1:5:end);
   Ucts=Ucts(1:5:end,:);
end
%{
```

Plot the simulation.

```
%}
figure(k),clf
xs=patches.x; xs([1 end],:)=nan;
surf(ts,xs(patches.hPts),Ucts(:,patches.hPts)'),hold on
surf(ts,xs(patches.uPts),Ucts(:,patches.uPts)'),hold off
xlabel('time t'),ylabel('space x'),zlabel('u(x,t) and h(x,t)')
view(70,45)
%{
```

Print the graph.

```
%}
if k==1, print('-depsc2','ps1WaveCtsUH')
else print('-depsc2','ps1WaterWaveCtsUH')
end
%{
```

Now, change to the Smagorinski turbulence model (2) of shallow water flow, keeping other parameters and the initial condition the same. And end the loop to redo the simulation.

```
%}
patches.fun=@waterWavepde;
dUdt0=patchSmooth1(0,U0(:));%check
end
%{
```

Use projective integration As yet a simple implementation appears to fail, so it needs more exploration and thought.

End the main function

```
%}
end
%{
```

3.5.1 Simple wave PDE

This function codes the staggered lattice equation inside the patches for the simple wave PDE system $h_t = -u_x$ and $u_t = -h_x$. Here code for a staggered microscale grid of staggered macroscale patches: the array

$$U_{ij} = \begin{cases} u_{ij} & i+j \text{ even}, \\ h_{ij} & i+j \text{ odd}. \end{cases}$$

The output `Ut` contains the merged time derivatives of the two staggered fields. So set the micro-grid spacing and reserve space for time derivatives.

```
%}
function Ut=simpleWavepde(t,U,x)
global patches
dx=x(2)-x(1);
Ut=nan(size(U));
ht=Ut;
%{
```

Compute the PDE derivatives at points internal to the patches.

```
%}
i=2:size(U,1)-1;
%{
```

Here 'wastefully' compute time derivatives for both PDEs at all grid points—for 'simplicity'—and then merges the staggered results. Since $\dot h_{ij} \approx -(u_{i+1,j} - u_{i-1,j})/(2\cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2\cdot dx)$ as adding/subtracting one from the index of a h-value is the location of the neighbouring u-value on the staggered micro-grid.

```
%}
ht(i,:)=-(U(i+1,:)-U(i-1,:))/(2*dx);
%{
```

Since $\dot u_{ij} \approx -(h_{i+1,j} - h_{i-1,j})/(2\cdot dx) = -(U_{i+1,j} - U_{i-1,j})/(2\cdot dx)$ as adding/subtracting one from the index of a u-value is the location of the neighbouring h-value on the staggered micro-grid.

```
%}
Ut(i,:)=-(U(i+1,:)-U(i-1,:))/(2*dx);
%{
```

Then overwrite the unwanted $\dot u_{ij}$ with the corresponding wanted $\dot h_{ij}$.

```
%}
Ut(patches.hPts)=ht(patches.hPts);
```

```
end
%{
```

3.5.2 Water wave PDE

This function codes the staggered lattice equation inside the patches for the nonlinear wave-like PDE system (2). As before, set the micro-grid spacing, reserve space for time derivatives, and index the internal points of the micro-grid.

```
%}
function Ut=waterWavepde(t,U,x)
global patches
dx=x(2)-x(1);
Ut=nan(size(U));
ht=Ut;
i=2:size(U,1)-1;
%{
```

Need to estimate h at all the u-points, so into `V` use averages, and linear extrapolation to patch-edges.

```
%}
ii=i(2:end-1);
V=Ut;
V(ii,:)=(U(ii+1,:)+U(ii-1,:))/2;
V(1:2,:)=2*U(2:3,:)-V(3:4,:);
V(end-1:end,:)=2*U(end-2:end-1,:)-V(end-3:end-2,:);
%{
```

Then estimate $\partial hu/\partial x$ from u and the interpolated h at the neighbouring micro-grid points.

```
%}
ht(i,:)=-(U(i+1,:).*V(i+1,:)-U(i-1,:).*V(i+1,:))/(2*dx);
%{
```

Correspondingly estimate the terms in the momentum PDE: u-values in `U`$_i$ and `V`$_{i\pm1}$; and h-values in `V`$_i$ and `U`$_{i\pm1}$.

```
%}
Ut(i,:)=-0.985*(U(i+1,:)-U(i-1,:))/(2*dx) ...
    -0.003*U(i,:).*abs(U(i,:)./V(i,:)) ...
    -1.045*U(i,:).*(V(i+1,:)-V(i-1,:))/(2*dx) ...
    +0.26*abs(V(i,:).*U(i,:)).*(V(i+1,:)-2*U(i,:)+V(i-1,:))/dx^2/2;
%{
```

where the mysterious division by two in the 2nd derivative is due to using the averaged values of u in the estimate:

$$\begin{aligned} u_{xx} &\approx \frac{1}{4\delta^2}(u_{i-2} - 2u_i + u_{i+2}) \\ &= \frac{1}{4\delta^2}(u_{i-2} + u_i - 4u_i + u_i + u_{i+2}) \\ &= \frac{1}{2\delta^2}\left(\frac{u_{i-2}+u_i}{2} - 2u_i + \frac{u_i+u_{i+2}}{2}\right) \\ &= \frac{1}{2\delta^2}\left(\bar u_{i-1} - 2u_i + \bar u_{i+1}\right). \end{aligned}$$

Then overwrite the unwanted $\dot u_{ij}$ with the corresponding wanted $\dot h_{ij}$.

```
%}
Ut(patches.hPts)=ht(patches.hPts);
end
%{
```

Fin.

3.6 To do

- Testing is so far only qualitative. Need to be quantitative.
- Multiple space dimensions.
- Heterogeneous microscale via averaging regions.

- Parallel processing versions.
- ??
- Adapt to maps in micro-time?

A Aspects of developing a ‘toolbox’ for patch dynamics

Section contents

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

A.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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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.

A.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)

A.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. E., Roberts, A. J. & Kevrekidis, I. G. (2017), 'Good coupling for the multiscale patch scheme on systems with microscale heterogeneity', *J. Computational Physics* **accepted 2 Feb 2017**.

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.

Cao, M. & Roberts, A. J. (2012), Modelling 3d turbulent floods based upon the smagorinski large eddy closure, *in* P. A. Brandner & B. W. Pearce, eds, '18th Australasian Fluid Mechanics Conference'.
http://people.eng.unimelb.edu.au/imarusic/proceedings/18/70%20-%20Cao.pdf

Cao, M. & Roberts, A. J. (2016*a*), 'Modelling suspended sediment in environmental turbulent fluids', *J. Engrg. Maths* **98**(1), 187–204.

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

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.

Kutz, J. N., Brunton, S. L., Brunton, B. W. & Proctor, J. L. (2016), *Dynamic Mode Decomposition: Data-driven Modeling of Complex Systems*, number 149 *in* 'Other titles in applied mathematics', SIAM, Philadelphia.

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.