Equation-Free function toolbox for Matlab/Octave

A. J. Roberts* et al.†

October 29, 2017

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

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.

Developers 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: Subsection 4.1 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:...}
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:...}
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

This is a very first stab at a good projective integration function.

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).

```
1 %}
2 function [xs,xss,tss]=projInt1(fun,x0,Ts,rank,dt,nTimeSteps)
3 %{
```

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.
- `nTimeSteps` is a two element vector:
  - `nTimeSteps(1)` is the number of microscale time-steps `dt` thought to be needed for microscale simulation to reach the slow manifold;

– `nTimeSteps(2)`, must be bigger than `rank`, is the number of subsequent time-steps to take which DMD analyses to mode the slow manifold.

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));
%{
```

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.

```
%}
x=[x0(:) nan(n,sum(nTimeSteps))];
for i=1:sum(nTimeSteps)
    xh=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,xh);
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(nTimeSteps))*dt nan];
end,end
%{
```

Grossly check on whether the microscale integration is stable. Is this any use??

```
%}
if norm(x(j,nTimeSteps(1)+(1:nTimeSteps(2)))) ...
  > 3*norm(x(j,1:nTimeSteps(1)))
  xMicroscaleIntegration=x, macroTime=Ts(k)
  error('projInt1: microscale integration appears unstable')
end
%{
```

DMD extrapolation over the macroscale DMD appears to work better when ones are adjoined to the data vectors for some unknown reason.

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

Then the basic DMD algorithm: first the fit.

```
%}
[U,S,V]=svd(x(j1,iStart:iFin-1),'econ');
S=diag(S);
Sr = S(1:rank) % rx1
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
x0(j)=Phi(1:end-1,:)*(bFin.*exp(omega*(DT(k)-iFin*dt))); % nx1
%{
```

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 `projInt1Test1`: A first test of basic projective integration

Seek to simulate the nonlinear diffusion PDE

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

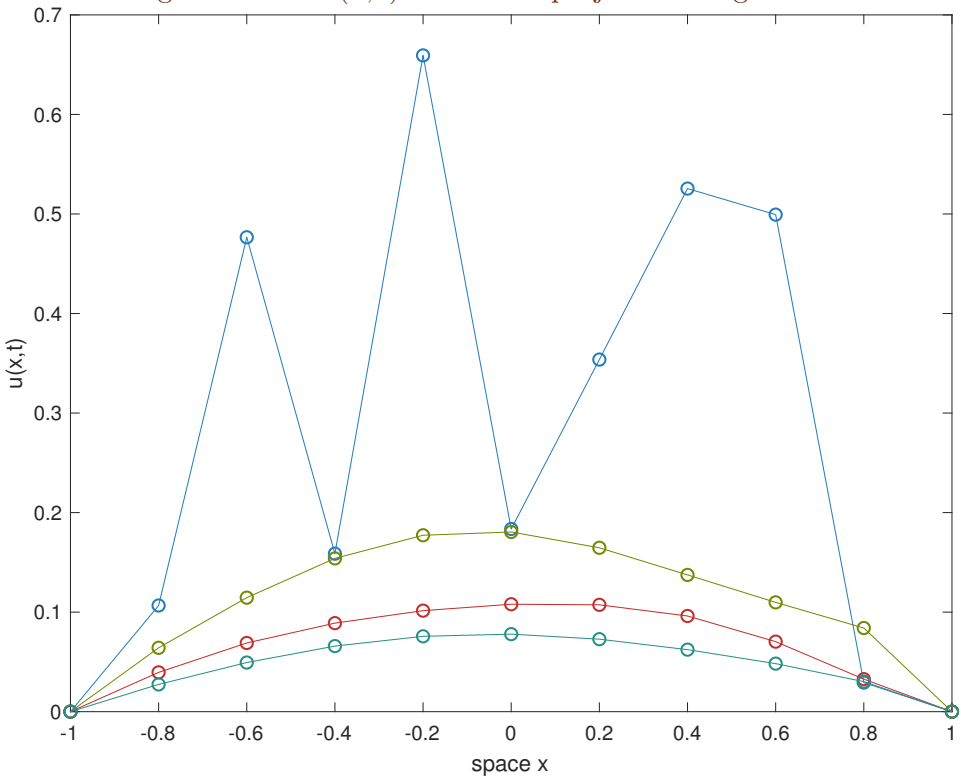
with random positive initial condition. Figure 1 shows solutions are attracted to the parabolic $u = a(t)(1 - x^2)$ with slow algebraic decay $\dot{a} = -2a^2$.

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

```
%}
function projInt1Test1
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.

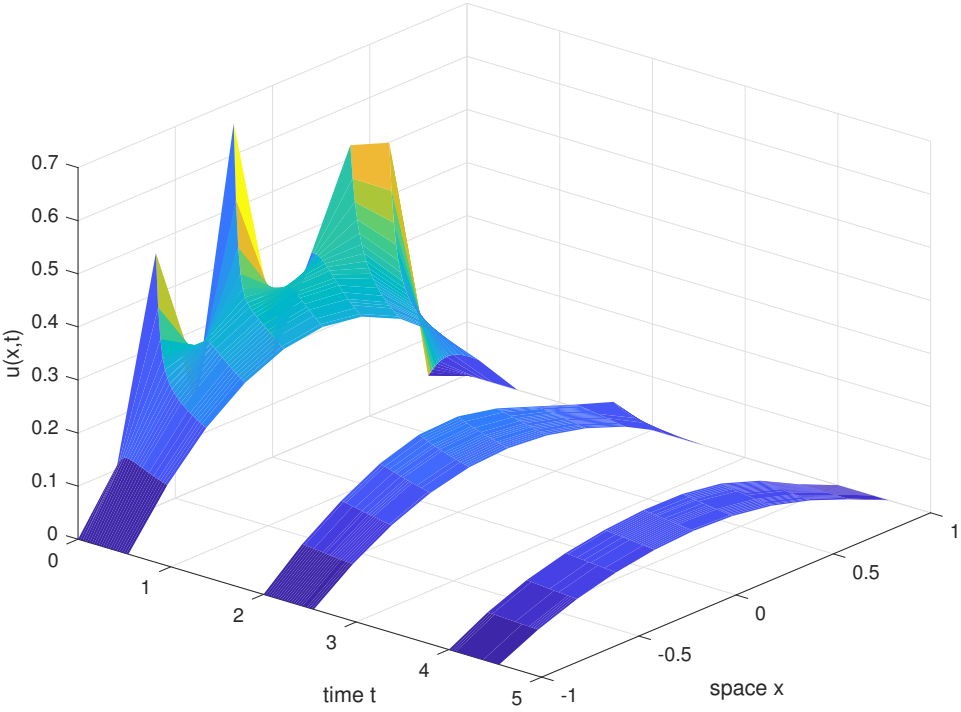



```
%}
x=linspace(-1,1,n+2)';
%u0=(1-x.^2).*(1+1e-9*randn(n+2,1));
u0=rand(n+2,1).*(1-x.^2);
%{
```

Projectively integrate in time one step for the moment with: rank-two DMD projection; guessed microscale time-step; and guessed numbers of transient (15) and slow (7) steps.

```
%}
[us,uss,tss]=projInt1(@dudt,u0,ts,2,2/n^2,[round(n^2/5.4) 7])
```

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




```
%{
```

Plot the macroscale predictions to draw Figure 1.

```
%}
clf,plot(x,us,'o-')
xlabel('space x'),ylabel('u(x,t)')
%matlab2tikz('pi1Test1u.ltx','noSize',true)
%print('-depsc2',['pi1Test1u' 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',['pi1Test1micro' 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.

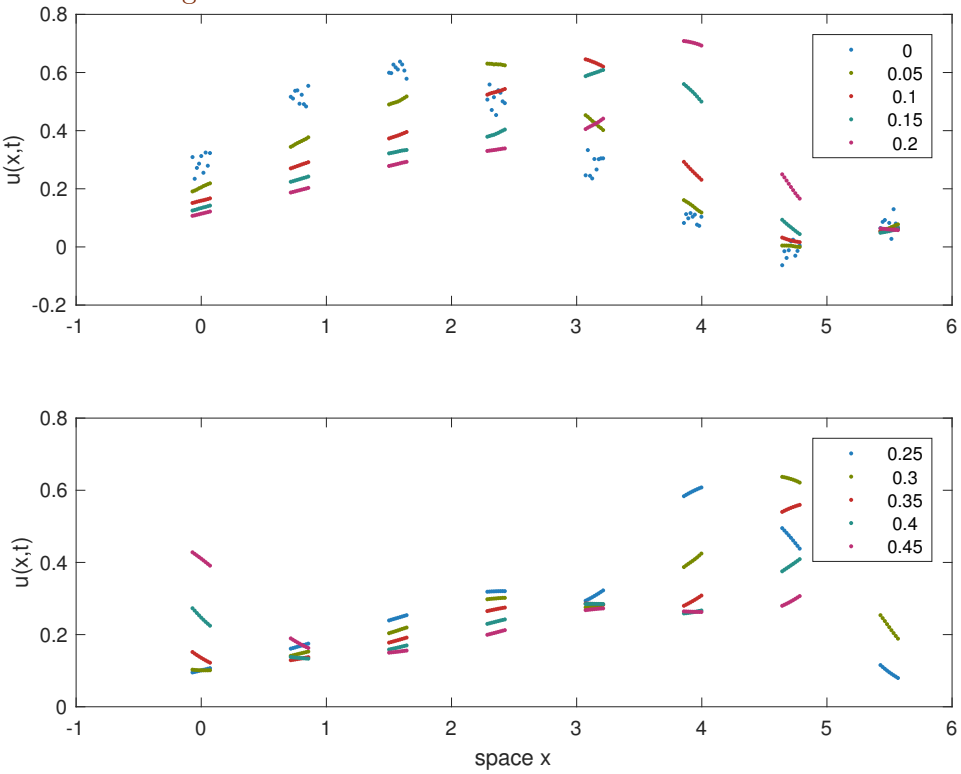
```
%}
function ut=dudt(t,u)
n=length(u);
dx=2/(n-1);
j=2:n-1;
ut=[0
    u(j).*(u(j+1)-2*u(j)+u(j-1))/dx^2
    0];
end
%{
```

2.3 `projInt1Patches`: Projective integration of patch scheme

Seek to simulate the nonlinear Burgers' PDE

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

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



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]);
%{
```

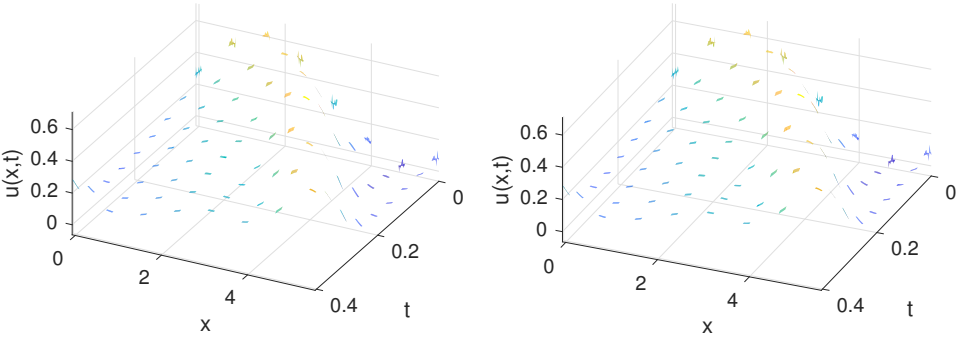
Plot the macroscale predictions to draw Figure 3, in groups of five in a plot.

```
%}
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')
%{
```

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

```
%}
tss(end)=nan; %omit end time-point
```

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




```
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 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

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 size `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.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`. The time derivatives must be computed into the same sized array, but the patch edge values will be overwritten by zeros.
- `patches.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.

Output

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

Try to figure out sizes of things. An error 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 conditions on the patch edge, 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 `NaN`s, 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. 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. For the moment?? assumes the domain is macro-periodic.

```
%}
dmu=(u(i0,jp,:)-u(i0,jm,:))/2; % \mu\delta
ddu=(u(i0,jp,:)-2*u(i0,j,:)+u(i0,jm,:)); % \delta^2
dddmu=dmu(1,jp,:)-2*dmu(1,j,:)+dmu(1,jm,:); % \mu\delta^3
ddddu=ddu(1,jp,:)-2*ddu(1,j,:)+ddu(1,jm,:); % \delta^4
%dddddmu=dddmu(jp,:)-2*dddmu(j,:)+dddmu(jm,:); % \mu\delta^5
%ddddddu=ddddu(jp,:)-2*ddddu(j,:)+ddddu(jm,:); % \delta^6
%{
```

Interpolate macro-values to be Dirichlet edge values for each patch (Roberts & Kevrekidis 2007). Here interpolate to fourth order.

```
%}
u(nM,j,:)=u(i0,j,:)+r*dmu+r^2/2*ddu ...
   +dddmu*(-1+r^2)*r/6+ddddu*(-1+r^2)*r^2/24;
u(1,j,:)=u(i0,j,:)-r*dmu+r^2/2*ddu ...
   -dddmu*(-1+r^2)*r/6+ddddu*(-1+r^2)*r^2/24;
%{
```

The following additions, respectively, is potentially sixth order.

```
   +dddddmu*(r/30-r^3/24+r^5/120) ...
   +ddddddu*(r^2/180-r^4/144+r^6/720);
   -dddddmu*(r/30-r^3/24+r^5/120) ...
   +ddddddu*(r^2/180-r^4/144+r^6/720);
```

Ask the user for the time derivatives, overwrite edge values, the return 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

Constructs the struct patches for use by the patch scheme patchDt.

```
%}
function makePatches(fun,Xa,Xb,nPatch,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.
- 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.x is nSubP $\times$ nPatch array of the regular spatial locations x_{ij} of the microscale grid points in every patch.

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

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

Second, 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`.

```
%}
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
%{
```

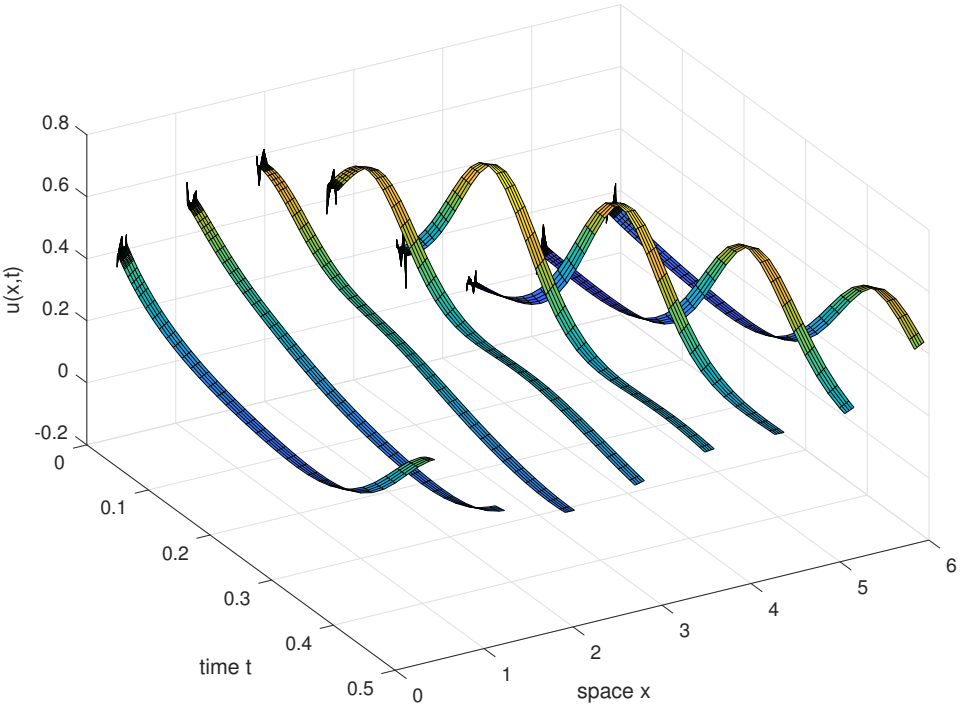
Fin.

3.3 BurgersTest: simulate Burgers' PDE on patches

Figure 5 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 BurgersTest
%{
```

Figure 5: field $u(x,t)$ tests the patch scheme function applied to Burgers' PDE.



Establish global data struct for Burgers' PDE solved on 2π-periodic domain, with eight patches, each patch of half-size 0.1, and with eleven points within each patch.

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

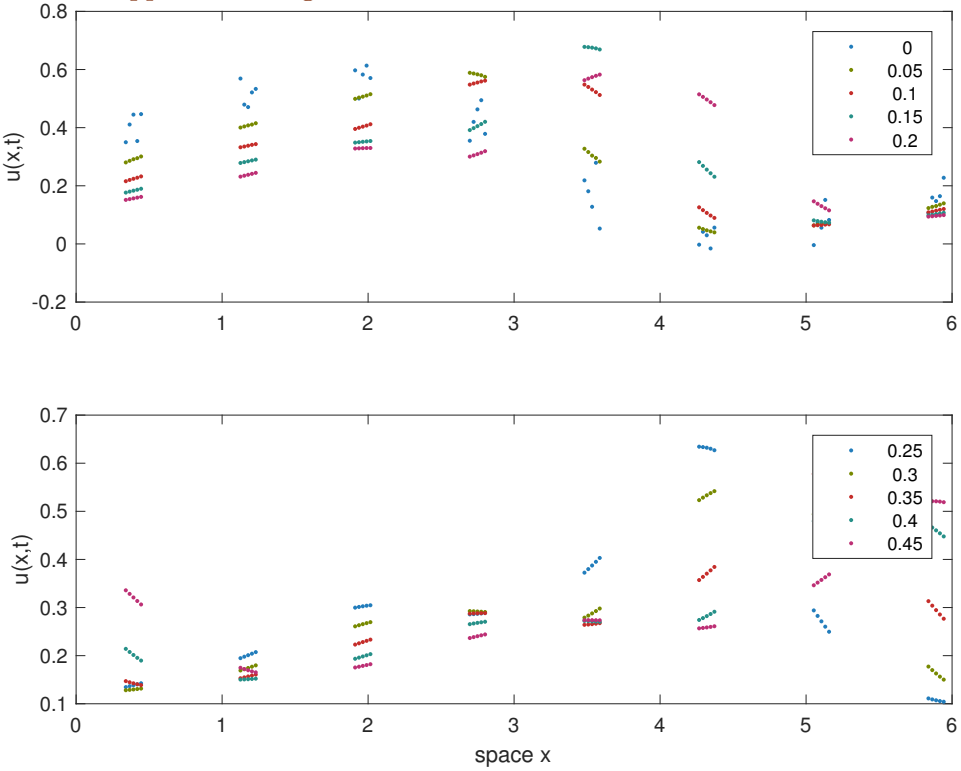
Plot the simulation.

```
%}
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 6.

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

Figure 6: field $u(x,t)$ tests basic projective integration of the patch scheme function applied to Burgers' PDE.




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

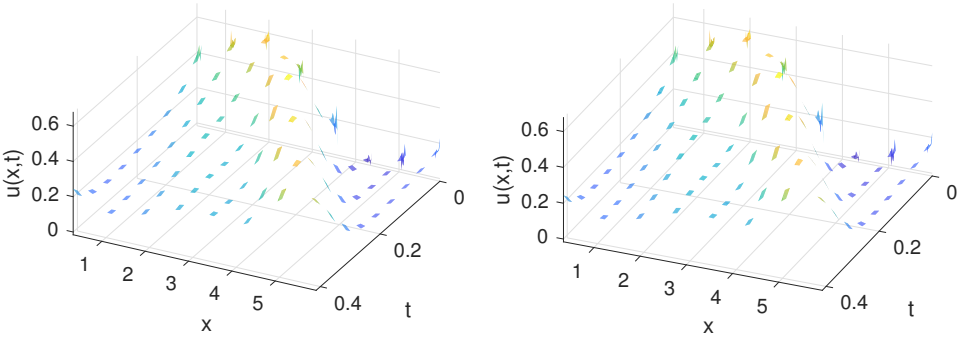
Plot the macroscale predictions to draw Figure 6, in groups of five in a plot.

```
%}
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)),'.')
  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 7.

```
%}
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')
```

Figure 7: stereo pair of the field $u(x,t)$ during each of the microscale bursts used in the projective integration.




```
%{
```

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 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?

4 Runge–Kutta 2 integration of deterministic ODEs

This describes a simple RK2 integration function, and some tests, as an example of the layout of information.

4.1 rk2int()

This is a simple example of Runge–Kutta, 2nd order, integration of a given deterministic ODE.

```
%}
function [xs,errs]=rk2int(fun,x0,ts)
%{
```

Input

- `fun()` is a function such as `dxdt=fun(x,t)` that computes the right-hand side of the ODE $d\vec{x}/dt = \vec{f}(\vec{x},t)$ 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 $\mathbb{R}^n$ vector of initial values at the time `ts(1)`.
- `ts` is a vector of times to compute the approximate solution, say in $\mathbb{R}^\ell$ for $\ell \geq 2$.

Output

- `xs`, array in $\mathbb{R}^{n\times\ell}$ of approximate solution vector at the specified times—but the transpose of what MATLAB does!
- `errs`, vector in $\mathbb{R}^{1\times\ell}$ of error estimate for the step from t_{k-1} to t_k.

Compute the time steps and create storage for outputs.

```
%}
dt=diff(ts);
```

```
xs=nan(length(x0),length(ts));
errs=nan(1,length(ts));
%{
```

Initialise first result to the given initial condition, and evaluate the initial time derivative into `f1`.

```
%}
xs(:,1)=x0(:);
errs(1)=0;
f1=fun(xs(:,1),ts(1));
%{
```

Compute the time-steps from t_k to t_{k+1}, copying the derivative `f1` at the end of the last time-step to be the derivative at the start of this one.

```
%}
for k=1:length(dt)
  f0=f1;
%{
```

Simple second order accurate time step.

```
%}
  xh=xs(:,k)+f0*dt(k)/2;
  fh=fun(xh,ts(k)+dt(k)/2);
  xs(:,k+1)=xs(:,k)+fh*dt(k);
  f1=fun(xs(:,k+1),ts(k+1));
%{
```

Use the time derivative at t_{k+1} to estimate an error by storing the difference with what Simpson's rule would estimate.

```
%}
  errs(k+1)=norm(f0-2*fh+f1)*dt(k)/6;
end
%{
```

End of the function with results returned in `xs` and `errs`.

4.2 rk2intTest1: A 1D test of RK2 integration

Try the nonlinear, but separable, ODE

$$\dot{x} = -2x/t \quad \Longrightarrow \quad x = c/t^2.$$

```
%}
fn=@(x,t) [-2*x/t]
%{
```

Solve ODE over $1 \le t \le 4$ with initial condition $x(1) = 1$

```
%}
ts=linspace(1,4,21);
[xs,errs]=rk2int(fn,1,ts)
plot(ts,xs,ts,100*errs,'o',ts,100*(xs-1 ./ts.^2),'x')
print -depsc2 RKInt/rk2intPlot1.eps
%{
```

Figure 8 shows the step-errors would accumulate to a reasonably conservative estimate of the solution error.

4.3 rk2intTest2: A 2D test of RK2 integration

Try the nonlinear fast-slow system

$$\dot{x} = -xy\,, \quad \dot{y} = -y + x^2.$$

```
%}
fn=@(x,t) [-x(1)*x(2);-x(2)+x(1)^2]
%{
```

Solve over $0 \le t \le 5$ with initial condition way off the slow manifold of $x(0) = 1$ and $y(0) = -1$.

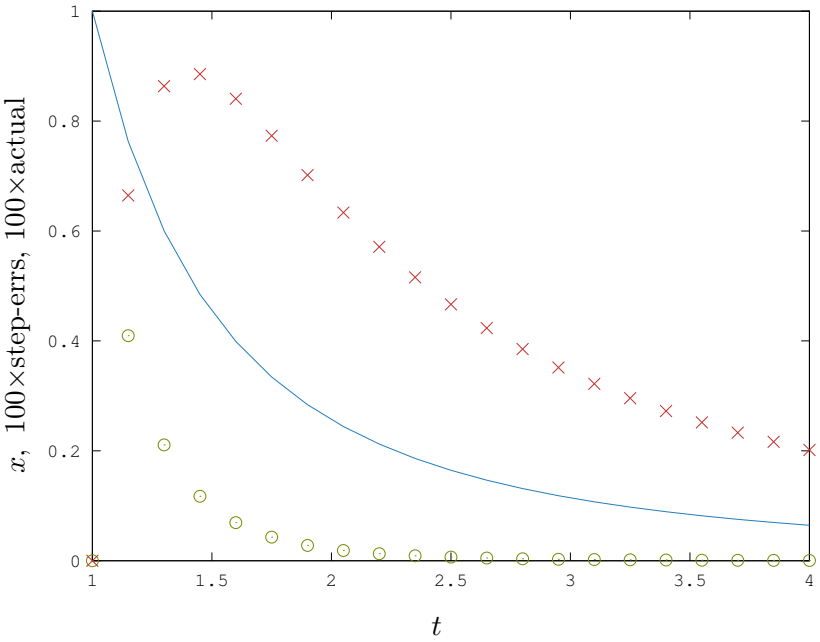


Figure 8: example trajectory and errors

```
%}
ts=linspace(0,5,21);
[xs,errs]=rk2int(fn,[1;-1],ts)
plot(ts,xs,ts,100*errs,'o')
print -depsc2 RKInt/rk2intPlot2.eps
%{
```

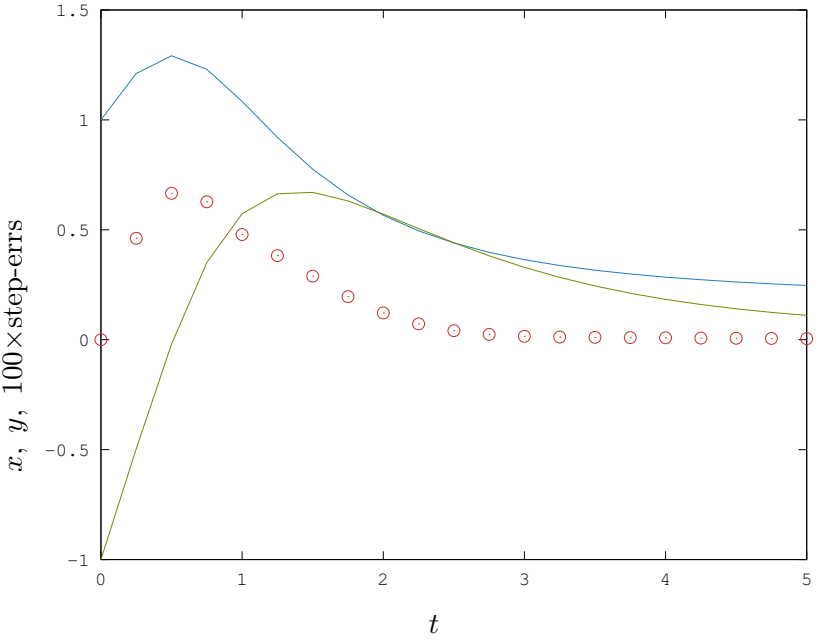


Figure 9: example trajectory and errors

A Aspects of developing a 'toolbox' for patch dynamics

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.

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., 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.

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

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.

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., 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.

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.