

# First draft Moving Mesh documentation

AJR

August 26, 2021

## Contents

<b>1</b>	<b>mm1dBurgersExample: example of moving patches for Burgers' PDE</b>	<b>1</b>
1.1	mmBurgersPDE(): Burgers PDE inside a moving mesh of patches	5
<b>2</b>	<b>mmPatchSys1(): interface 1D space of moving patches to time integrators</b>	<b>7</b>
<b>3</b>	<b>mm2dExample: example of moving patches in 2D for nonlinear diffusion</b>	<b>10</b>
3.1	mmNonDiffPDE(): nonlinear diffusion PDE inside moving patches	15
<b>4</b>	<b>mmPatchSys2(): interface 2D space of moving patches to time integrators</b>	<b>16</b>
<b>1</b>	<b>mm1dBurgersExample: example of moving patches for Burgers' PDE</b>	

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

1. configPatches1
2. ode15s integrator  $\leftrightarrow$  mmPatchSys1  $\leftrightarrow$  user's PDE
3. process results

The simulation seems perfectly happy for the patches to move so that they overlap in the shock! and then separate again as the shock decays.

Establish global patch data struct to point to and interface with a function coding Burgers' PDE: to be solved on 1-periodic domain, with fifteen patches,

spectral interpolation couples the patches, each patch of half-size ratio 0.2, and with five microscale points forming each patch. Prefer EdgyInt as we suspect it performs better for moving meshes.

```

31 clear all
32 global patches
33 patches = configPatches1(@mmBurgersPDE,[0 1], nan, 15, 0, 0.2, 5 ...
34     , 'EdgyInt', true);
35 patches.mmTime=1;
36 patches.Xlim=[0 1];

```

The above two amendments to `patches` should eventually be part of the configuration function.

**Decide the moving mesh time parameter** Here for  $\epsilon = 0.02$ .

- Would be best if the moving mesh was no stiffer than the stiffest microscale sub-patch mode. These would both be the zig-zag modes.
  - Here the mesh PDE is  $X_t = (N^2/\tau)X_{jj}$  so its zig-zag mode decays with rate  $4N^2/\tau$ .
  - Here the patch width is  $h = 0.2/15 = 1/75$ , and so the microscale step is  $\delta = h/4 = 1/300$ . Hence the diffusion  $u_t = \epsilon u_{xx}$  has zig-zag mode decaying at rate  $4\epsilon/\delta^2$ .

So, surely best to have  $4N^2/\tau \lesssim 4\epsilon/\delta^2$ , that is,  $\tau \gtrsim N^2\delta^2/\epsilon \approx 0.1$ .

- But also we do not want the slowest modes of the moving mesh to obfuscate the system's macroscale modes—the macroscale zig-zag.
  - The slowest moving mesh mode has wavenumber in  $j$  of  $2\pi/N$ , and hence rate of decay  $(N^2/\tau)(2\pi/N)^2 = 4\pi^2/\tau$ .
  - The fastest zig-zag mode of the system  $U_t = \epsilon U_{xx}$  on step  $H$  has decay rate  $4\epsilon/H^2$ .

So best if  $4\pi^2/\tau \gtrsim 4\epsilon/H^2$ , that is,  $\tau \lesssim \pi^2 H^2/\epsilon \approx 2$ .

(Computations indicate need  $\tau < 0.8??$ )

**Simulate in time** Set usual sinusoidal initial condition. Add some microscale randomness that decays within time of 0.01, but also seeds slight macroscale variations.

```
83 u0 = 0.3+sin(2*pi*patches.x)+0.05*randn(size(patches.x));
84 N = size(patches.x,4)
85 D0 = zeros(N,1);
```

Simulate in time using a standard stiff integrator and the interface function `mmPatchsmooth1()` ([Section 2](#)).

```
93 tic
94 [ts,us] = ode15s(@mmPatchSys1,linspace(0,0.8),[D0;u0(:)]);
95 cpuTime = toc
```

**Plots** Choose whether to save some plots, or not.

```
104 global OurCf2eps
105 OurCf2eps = false;
```

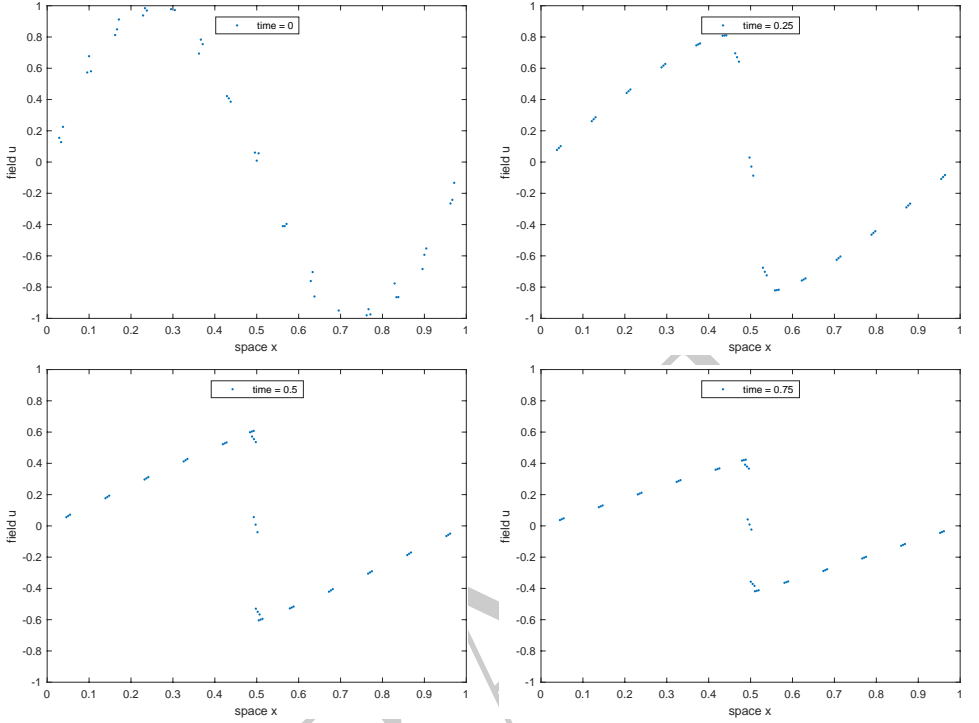
Plot the movement of the mesh, the centre of each patch, as a function of time: spatial domain horizontal, and time vertical.

```
114 figure(1),clf
115 Ds=us(:,1:N);
116 Xs=shiftdim(mean(patches.x),2);
117 plot(Xs+Ds,ts), ylabel('time t'),xlabel('space x')
118 title('Burgers PDE: patch locations over time')
```

Animate the simulation using only the microscale values interior to the patches: set  $x$ -edges to `nan` to leave the gaps. [Figure 1](#) illustrates an example simulation in time generated by the patch scheme applied to Burgers' PDE.

```
129 us=us(:,N+1:end).';
130 us(abs(us)>2)=nan;
131 x0s=squeeze(patches.x); x0s([1 end],:)=nan;
132 %% section break to ease rerun of animation
133 figure(2),clf
134 for i=1:length(ts)
135     xs=x0s+Ds(i,:);
136     if i==1, hpts=plot(xs(:),us(:,i),'.');
137         ylabel('field u'), xlabel('space x')
```

Figure 1: field  $u(x,t)$  of the moving patch scheme applied to Burgers' PDE.



```

138     axis([0 1 -1 1])
139     else set(hpts,'XData',xs(:),'YData',us(:,i));
140     end
141     legend(['time = ' num2str(ts(i),2)],'Location','north')
142     if rem(i,31)==1, if0urCf2eps([mfilename num2str(i)]), end
143     pause(0.04)
144 end
145 %%

```

**Spectrum of the moving patch system** Compute the spectrum based upon the linearisation about some state:  $u = \text{constant}$  with  $D = 0$  are equilibria; otherwise the computation is about a 'quasi-equilibrium' on the 'fast-time'.

```

169 u0 = 0.1+0*sin(2*pi*patches.x);
170 u0 = [zeros(N,1); u0(:)];

```

```

171 f0 = mmPatchSys1(0,u0);
172 normf0=norm(f0)

```

But we must only use the dynamic variables, so let's find where they are.

```

179 xs=patches.x; xs([1 end],:,:,:) = nan;
180 i=find(~isnan( [zeros(N,1);xs(:)] ));
181 nJac=length(i)

```

Construct Jacobian with numerical differentiation.

```

187 deltau=1e-7;
188 Jac=nan(nJac);
189 for j=1:nJac
190     uj=u0; uj(i(j))=uj(i(j))+deltau;
191     fj = mmPatchSys1(0,uj);
192     Jac(:,j)=(fj(i)-f0(i))/deltau;
193 end

```

Compute and plot the spectrum with non-linear axis scaling ([Figure 2](#)).

```

200 eval=-sort(-eig(Jac))
201 figure(3),clf
202 plot(asinh(real(eval)),asinh(imag(eval)),'.')
203 xlabel('Re\lambda'), ylabel('Im\lambda')
204 ticks=[1;2;5]*10.^(0:4);
205 ticks=sort([0;ticks(:);-ticks(:)]);
206 set(gca,'Xtick',asinh(ticks) ...
207     , 'XtickLabel',cellstr(num2str(ticks,4)) ...
208     , 'XTickLabelRotation',30)
209 set(gca,'Ytick',asinh(ticks) ...
210     , 'YtickLabel',cellstr(num2str(ticks,4)))
211 grid
212 if0urCf2eps([mfilename 'Spec'])

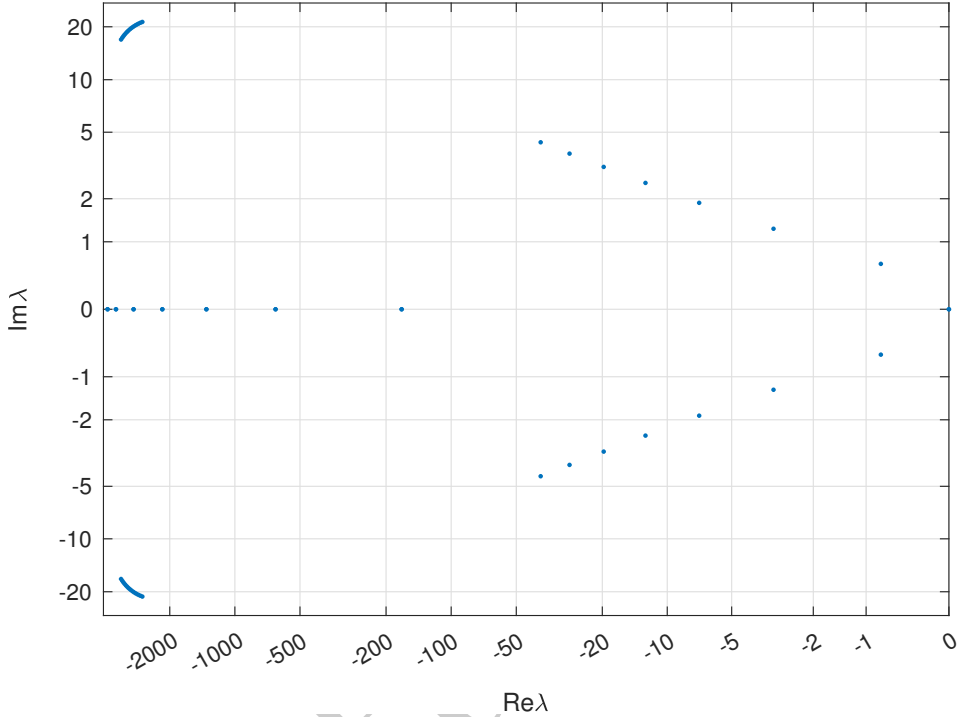
```

Fin.

## 1.1 mmBurgersPDE(): Burgers PDE inside a moving mesh of patches

For the evolving scalar field  $u(t, x)$ , we code a microscale discretisation of Burgers' PDE  $u_t = \epsilon u_{xx} - uu_x$ , for say  $\epsilon = 0.02$ , when the patches of microscale lattice move with various velocities  $V$ .

Figure 2: spectrum of the moving mesh Burgers' system (about  $u = 0.1$ ). The four clusters are: right, macroscale Burgers' PDE (complex conjugate pairs); left complex pairs, sub-patch PDE modes; left real, moving mesh modes.



```

15 function ut = mmBurgersPDE(t,u,M,patches)
16 epsilon = 0.02;

```

### Generic input/output variables

- $\mathbf{t}$  (scalar) current time—not used here as the PDE has no explicit time dependence (autonomous).
- $\mathbf{u}$  ( $n \times 1 \times 1 \times N$ ) field values on the patches of microscale lattice.
- $\mathbf{M}$  a struct of the following components.
  - $\mathbf{V}$  ( $1 \times 1 \times 1 \times N$ ) moving velocity of the  $j$ th patch.
  - $\mathbf{D}$  ( $1 \times 1 \times 1 \times N$ ) displacement of the  $j$ th patch from the fixed spatial positions stored in `patches.x`—not used here as the PDE has no explicit space dependence (homogeneous).

- `patches` struct of patch configuration information.
- `ut` ( $n \times 1 \times 1 \times N$ ) output computed values of the time derivatives  $Du/Dt$  on the patches of microscale lattice.

Here there is only one field variable, and one in the ensemble, so for simpler coding of the PDE we squeeze them out (no need to reshape when via `mmPatchSys1`).

```

47 u=squeeze(u);          % omit singleton dimensions
48 V=shiftdim(M.V,2); % omit two singleton dims

```

**Burgers PDE** In terms of the moving derivative  $Du/Dt := u_t + Vu_x$  the PDE becomes  $Du/Dt = \epsilon u_{xx} + (V - u)u_x$ . So code for every patch that  $\dot{u}_{ij} = \frac{\epsilon}{h^2}(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}) + (V_j - u_{ij})\frac{1}{2h}(u_{i+1,j} - u_{i-1,j})$  at all interior lattice points.

```

61 dx=diff(patches.x(1:2)); % microscale spacing
62 i=2:size(u,1)-1; % interior points in patches
63 ut=nan+u; % preallocate output array
64 ut(i,:) = epsilon*diff(u,2)/dx^2 ...
65         +(V-u(i,:)).*(u(i+1,:)-u(i-1,:))/(2*dx);
66 end

```

## 2 `mmPatchSys1()`: interface 1D space of moving patches to time integrators

To simulate in time with moving 1D spatial patches we need to interface a user's time derivative function with time integration routines such as `ode23` or `PIRK2`. This function `mmPatchSys1()` provides an interface. Patch edge values are determined by macroscale interpolation of the patch-centre or edge values. Microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (??) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```

28 function dudt = mmPatchSys1(t,u,patches)
29 if nargin<3, global patches, end

```

## Input

- **u** is a vector of length  $\mathbf{nPatch} + \mathbf{nSubP} \cdot \mathbf{nVars} \cdot \mathbf{nEnsem} \cdot \mathbf{nPatch}$  where there are  $\mathbf{nVars} \cdot \mathbf{nEnsem}$  field values at each of the points in the  $\mathbf{nSubP} \times \mathbf{nPatch}$  grid, and because of the moving mesh there are an additional  $\mathbf{nPatch}$  patch displacement values at its start.
- **t** is the current time to be passed to the user's time derivative function.
- **patches** a struct set by `configPatches1()` with the following information used here.
  - **.fun** is the name of the user's function `fun(t,u,M,patches)` that computes the time derivatives on the patchy lattice, where the  $j$ th patch moves at velocity  $M.V_j$  and at current time is displaced  $M.D_j$  from the fixed reference position in **.x**. The array **u** has size  $\mathbf{nSubP} \times \mathbf{nVars} \times \mathbf{nEnsem} \times \mathbf{nPatch}$ . Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - **.x** is  $\mathbf{nSubP} \times 1 \times 1 \times \mathbf{nPatch}$  array of the spatial locations  $x_i$  of the microscale grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and microscales ??

## Output

- **dudt** is a vector of of time derivatives, but with patch edge-values set to zero. It is of total length  $\mathbf{nPatch} + \mathbf{nSubP} \cdot \mathbf{nVars} \cdot \mathbf{nEnsem} \cdot \mathbf{nPatch}$ .

Extract the  $\mathbf{nPatch}$  displacement values from the start of the vectors of evolving variables. Reshape the rest as the fields **u** in a 4D-array, and sets the edge values from macroscale interpolation of centre-patch values. `??` describes `patchEdgeInt1()`.

```

90 N = size(patches.x,4);
91 M.D = reshape(u(1:N),[1 1 1 N]);
92 u = patchEdgeInt1(u(N+1:end),patches);

```

**Moving mesh velocity** Developing from standard moving meshes for PDES (Budd et al. 2009, Huang & Russell 2010, e.g.), we follow Maclean, Bunder, Kevrekidis & Roberts (2021). There exists a set of macro-scale mesh points  $X_j(t) := X_j^0 + D_j(t)$  (at the centre) of each patch with associated field values, say  $U_j(t) := \overline{u_{ij}(t)}$ .



```

106 X = mean(patches.x,1)+M.D;
107 U = mean(u,1);

```

Then for every patch  $j$  we set  $H_j := X_{j+1} - X_j$  for periodic patch indices  $j$

```

114 j=1:N; jp=[2:N 1]; jm=[N 1:N-1];
115 H = X(:,:, :, jp)-X(:,:, :, j);
116 H(N) = H(N)+diff(patches.Xlim);

```

we discretise a moving mesh PDE for node locations  $X_j$  with field values  $U_j$  via the second derivative estimate

$$U_j'' := \frac{2}{H_j + H_{j-1}} \left[ \frac{U_{j+1} - U_j}{H_j} - \frac{U_j - U_{j-1}}{H_{j-1}} \right], \quad (1a)$$

and its norm over all variables and ensembles (arbitrarily?? chose the mean square norm here).

```

130 U2 = ( (U(:,:, :, jp)-U(:,:, :, j))./H(:,:, :, j) ...
131         -(U(:,:, :, j)-U(:,:, :, jm))./H(:,:, :, jm) ...
132         )*2./(H(:,:, :, j)+H(:,:, :, jm)));
133 U2 = squeeze( mean(mean( abs(U2).^2 ,2),3) );
134 H = squeeze(H);

```

Having squeezed out all microscale information, the coefficient

$$\alpha := \max \left\{ 1, \left[ \frac{1}{b-a} \sum_j H_{j-1} \frac{1}{2} \left( U_j''^{2/3} + U_{j-1}''^{2/3} \right) \right]^3 \right\} \quad (1b)$$

Rather than  $\max(1, \cdot)$  surely better to use something smooth like  $\sqrt[3]{1 + \cdot}$ ??

```

149 alpha = sum( H(jm).*( U2(j).^ (1/3)+U2(jm).^ (1/3) ))/sum(H);
150 alpha = max(1,alpha^3);

```

Then the importance function

$$\rho_j := \left( 1 + \frac{1}{\alpha} U_j''^2 \right)^{1/3}, \quad (1c)$$

```

160 rho = ( 1+U2/alpha ).^ (1/3);

```

For every patch, we move all micro-grid points according to the following velocity of the notional macro-scale node of that patch:

$$V_j := \frac{dX_j}{dt} = \frac{(N-1)^2}{2\rho_j\tau} [(\rho_{j+1} + \rho_j)H_j - (\rho_j + \rho_{j-1})H_{j-1}]. \quad (1d)$$

```

174 M.V = nan+M.D; % allocate storage
175 M.V(:) = ( (rho(jp)+rho(j)).*H(j) -(rho(j)+rho(jm)).*H(jm) ) ...
176 ./rho(j) *((N-1)^2/2/patches.mmTime);

```

**Evaluate system differential equation** Ask the user function for the advected time derivatives on the moving patches, overwrite its edge values with the dummy value of zero (since `ode15s` chokes on NaNs), then return to the user/integrator as a vector.

```

189 dudt=patches.fun(t,u,M,patches);
190 dudt([1 end],:,:) = 0;
191 dudt=[M.V(:); dudt(:)];

```

Fin.

### 3 mm2dExample: example of moving patches in 2D for nonlinear diffusion

The code here shows one way to use moving patches in 2D. However, `mmPatchSys2()` has far too many ad hoc assumptions, so fix those before exploring predictions here.

Establish global patch data struct to interface with a function coding a nonlinear ‘diffusion’ PDE: to be solved on  $6 \times 4$ -periodic domain, with  $9 \times 7$  patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.4?? (relatively large for visualisation), and with  $5 \times 5$  points forming each patch. [Roberts et al. \(2014\)](#) established that this scheme is consistent with the PDE (as the patch spacing decreases). Prefer `EdgyInt` as we suspect it performs better for moving meshes.

```

29 clear all
30 global patches
31 nxy=5
32 Nx=9, Ny=7
33 patches = configPatches2(@mmNonDiffPDE,[-3 3 -2 2], nan ...
34     , [Nx Ny], 0, 0.2, nxy , 'EdgyInt',true);
35 patches.mmTime=1;
36 patches.Xlim=[-3 3 -2 2];
37 Npts = Nx*Ny;

```

The above two amendments to `patches` should eventually be part of the configuration function.

## Decide the moving mesh time parameter

**Simulate in time** Set an initial condition of a perturbed-Gaussian using auto-replication of the spatial grid.

```
84 u0 = exp(-patches.x.^2-patches.y.^2);
85 u0 = u0.*(0.9+0.0*rand(size(u0))) +0.001;
86 D0 = zeros(2*Npts,1);
```

Integrate in time to  $t = 2$  using standard functions. In MATLAB `ode15s` would be natural as the patch scheme is naturally stiff, but `ode23` is quicker (Maclean, Bunder & Roberts 2021, Fig. 4). Ask for output at non-uniform times because the diffusion slows.

```
97 disp('Simulating nonlinear diffusion h_t=(h^3)_xx+(h^3)_yy')
98 tic
99 [ts,us] = ode23(@mmPatchSys2,2*linspace(0,1).^2,[D0;u0(:)]);
100 cpuTime = toc
```

**Plots** Choose whether to save some plots, or not.

```
108 global OurCf2eps
109 OurCf2eps = true;
```

Extract data from time simulation. Be wary that the patch-edge values do not change from initial, so either set to NaN, or set via interpolation.

```
115 nTime=length(ts);
116 Ds=reshape(us(:,1:2*Npts).',1,1,Nx,Ny,2,nTime);
117 us=reshape(us(:,2*Npts+1:end).',nxy,nxy,Nx,Ny,nTime);
118 us([1 end],:,:,:) = nan; us(:,[1 end],:,:) = nan; % nan edges
```

Choose macro-mesh plot or micro-surf-patch plots.

```
125 if 1
```

Plot the movement of the mesh, with the field vertical, at the centre of each patch.

```
132 % section marker for macro-mesh plot execution
133 figure(1),clf, colormap(0.8*hsv)
134 Us=shiftdim(mean(mean(us,1,'omitnan'),2,'omitnan'),2);
135 Xs=shiftdim(mean(patches.x),4);
```

```

136 Ys=shiftdim(mean(patches.y),4);
137 for k=1:nTime
138     Xk=Xs+shiftdim(Ds(:,:,:,:),1,k),2);
139     Yk=Ys+shiftdim(Ds(:,:,:,:),2,k),2);
140     if k==1,
141         hand=mesh(Xk,Yk,Us(:,:,k));
142         ylabel('space y'),xlabel('space x'),zlabel('mean field U')
143         axis([patches.Xlim 0 1]), caxis([0 1])
144         colorbar
145         if 0, view(0,90) % vertical view
146         else view(-25,60) % 3D perspective
147         end
148     else
149         set(hand,'XData',Xk,'YData',Yk ...
150             , 'ZData',Us(:,:,k),'CData',Us(:,:,k))
151     end
152     legend(['time =' num2str(ts(k),4)], 'Location', 'north')
153     if rem(k,31)==1, ifOurCf2eps([mfilename num2str(k)]), end
154     pause(0.05)
155 end% for each time
156 else%if macro-mesh or micro-surf

```

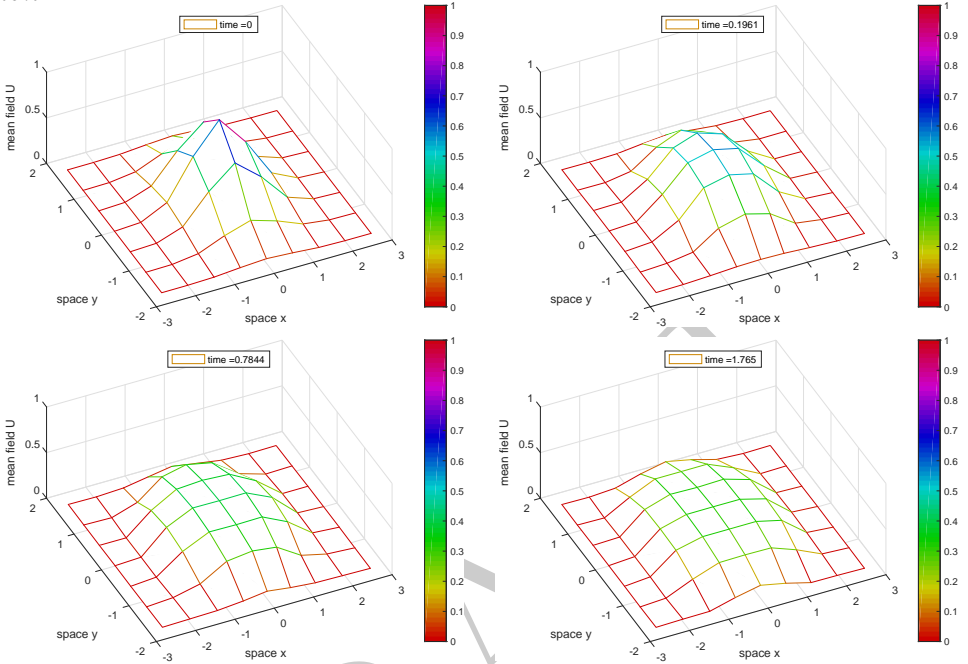
Plot the movement of the patches, with the field vertical in each patch.

```

163 %% section marker for patch-surf plot execution
164 figure(2),clf, colormap(0.8*hsv)
165 xs=reshape(patches.x,nxy,1,Nx,1);
166 ys=reshape(patches.y,1,nxy,1,Ny);
167 for k=1:nTime
168     xk=xs+0*ys+Ds(:,:,:,:),1,k);
169     yk=ys+0*xs+Ds(:,:,:,:),2,k);
170     uk=reshape(permute(us(:,:,:,:),[1 3 2 4]),nxy*Nx,nxy*Ny);
171     xk=reshape(permute(xk,[1 3 2 4]),nxy*Nx,nxy*Ny);
172     yk=reshape(permute(yk,[1 3 2 4]),nxy*Nx,nxy*Ny);
173     if k==1,
174         hand=surf(xk,yk,uk);
175         ylabel('space y'),xlabel('space x'),zlabel('field u(x,y,t)')
176         axis([patches.Xlim 0 1]), caxis([0 1])
177         colorbar
178     else

```

Figure 3: field  $u(x,t)$  of the moving patch scheme applied to nonlinear diffusion PDE.



```

179     set(hand,'XData',xk,'YData',yk,'ZData',uk,'CData',uk)
180     end
181     legend(['time = ' num2str(ts(k),4)],'Location','north')
182 % if rem(k,31)==1, ifOurCf2eps([mfilename num2str(k)]), end
183     pause(0.05)
184 end% for each time
185 %%
186 end%if macro-mesh or micro-surf

```

**Spectrum of the moving patch system** Compute the spectrum based upon the linearisation about some state:  $u = \text{constant}$  with  $D = 0$  are equilibria; otherwise the computation is about a 'quasi-equilibrium' on the 'fast-time'.

```

213 u00 = 0.1
214 u0 = u00+0.1*exp(-patches.x.^2-patches.y.^2);
215 u0([1 end],:,:,:) = nan; u0(:, [1 end],:,:) = nan;
216 u0 = [zeros(2*Npts,1); u0(:)];

```

```

217 f0 = mmPatchSys2(0,u0);
218 normf0=norm(f0)

```

But we must only use the dynamic variables, so let's find where they are.

```

225 i=find(~isnan( u0(:) ));
226 nJac=length(i)

```

Construct Jacobian with numerical differentiation.

```

232 deltau=1e-7;
233 Jac=nan(nJac);
234 for j=1:nJac
235     uj=u0; uj(i(j))=uj(i(j))+deltau;
236     fj = mmPatchSys2(0,uj);
237     Jac(:,j)=(fj(i)-f0(i))/deltau;
238 end

```

Compute and plot the spectrum with non-linear axis scaling ([Figure 4](#)).

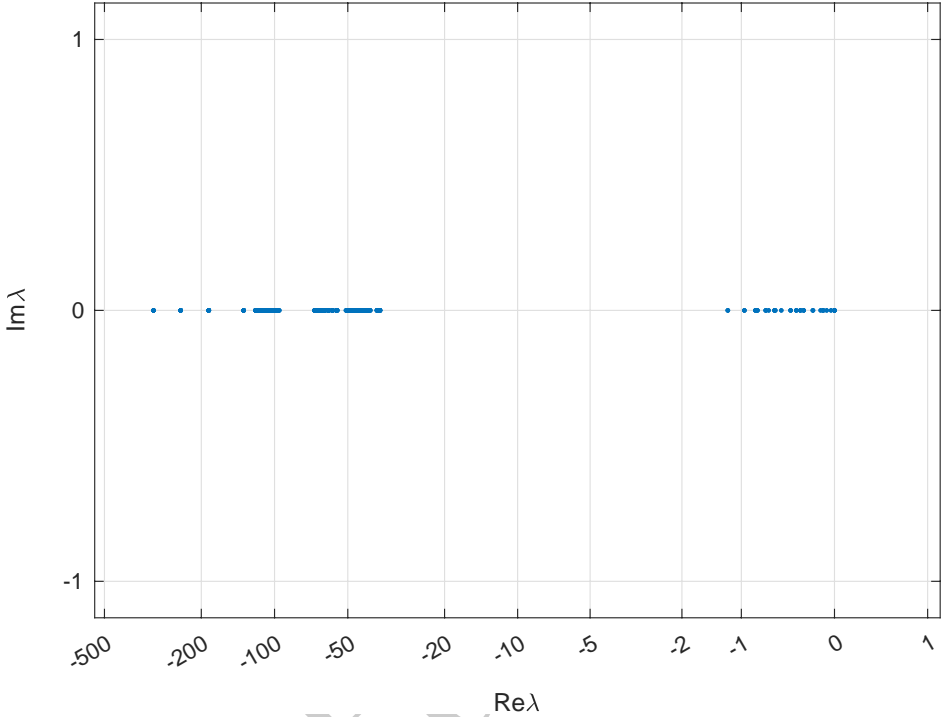
```

245 eval=eig(Jac);
246 k=find(abs(imag(eval))<1e-6);
247 eval(k)=real(eval(k));
248 [~,k]=sort(-real(eval));
249 eval=eval(k);
250 nZero = sum(abs(real(eval))<1e-6)
251 nSlow = sum(-3*u00^2*300<real(eval))-nZero
252 eSlow = eval(nZero+(1:2:nSlow))
253 eFast = eval([nZero+nSlow+1 end])
254 figure(3),clf
255 plot(asinh(real(eval)),asinh(imag(eval)),'.')
256 xlabel('Re\lambda'), ylabel('Im\lambda')
257 ticks=[1;2;5]*10.^(0:6);
258 ticks=sort([0;ticks(:);-ticks(:)]);
259 set(gca,'Xtick',asinh(ticks) ...
260     , 'XtickLabel',cellstr(num2str(ticks,4)) ...
261     , 'XTickLabelRotation',30)
262 set(gca,'Ytick',asinh(ticks) ...
263     , 'YtickLabel',cellstr(num2str(ticks,4)))
264 grid
265 ifOurCf2eps([mfilename 'Spec'])

```

Fin.

Figure 4: spectrum of the moving mesh 2D diffusion system (about  $u = 0.1$ ). The clusters are: right real, macroscale diffusion modes with some neutral mesh deformations; left real, moving mesh and sub-patch modes.



### 3.1 mmNonDiffPDE(): nonlinear diffusion PDE inside moving patches

As a microscale discretisation of  $u_t = \vec{V} \cdot \vec{\nabla} u + \nabla^2(u^3)$ , code  $u_{ijkl} = \dots + \frac{1}{\delta x^2}(u_{i+1,j,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i-1,j,k,l}^3) + \frac{1}{\delta y^2}(u_{i,j+1,k,l}^3 - 2u_{i,j,k,l}^3 + u_{i,j-1,k,l}^3)$ .

```

13 function ut = mmNonDiffPDE(t,u,M,patches)
14     if nargin<3, global patches, end
15     u = squeeze(u); % reduce to 4D
16     Vx = shiftdim(M.Vx,2); % omit two singleton dims
17     Vy = shiftdim(M.Vy,2); % omit two singleton dims
18     dx = diff(patches.x(1:2)); % microgrid spacing
19     dy = diff(patches.y(1:2));
20     i = 2:size(u,1)-1; j = 2:size(u,2)-1; % interior patch points
21     ut = nan+u; % preallocate output array

```

```

22   ut(i,j,::) = ...
23       +Vx.*(u(i+1,j,::)-u(i-1,j,::))/(2*dx) ...
24       +Vy.*(u(i,j+1,::)-u(i,j-1,::))/(2*dy) ...
25       +diff(u(:,j,::).^3,2,1)/dx^2 ...
26       +diff(u(i,::,::).^3,2,2)/dy^2 ;
27 end

```

## 4 mmPatchSys2(): interface 2D space of moving patches to time integrators

**Beware ad hoc assumptions** In an effort to get started, I have just made some plausible generalisations from the 1D code to this 2D code. Probably lots of details are poor??

To simulate in time with 2D patches moving in space we need to interface a users time derivative function with time integration routines such as `ode23` or `PIRK2`. This function `mmPatchSys2()` provides an interface. Patch edge values are determined by macroscale interpolation of the patch-centre or edge values. Microscale heterogeneous systems may be accurately simulated with this function via appropriate interpolation. Communicate patch-design variables (??) either via the global struct `patches` or via an optional third argument (except that this last is required for parallel computing of `spmd`).

```

30 function dudt = mmPatchSys2(t,u,patches)
31 if nargin<3, global patches, end

```

### Input

- `u` is a vector of length  $2 \cdot \text{prod}(\text{nPatch}) + \text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$  where there are  $\text{nVars} \cdot \text{nEnsem}$  field values at each of the points in the  $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nPatch}(1) \times \text{nPatch}(2)$  grid.
- `t` is the current time to be passed to the user's time derivative function.
- `patches` a struct set by `configPatches2()` with the following information used here.
  - `.fun` is the name of the user's function `fun(t,u,M,patches)` that computes the time derivatives on the patchy lattice, where the  $(I, J)$ th patch moves at velocity  $(M.V_{xI}, M.V_{yJ})$  and at current



time is displaced  $(M.Dx_I, M.Dy_J)$  from the fixed reference positions in `.x` and `.y`. The array `u` has size  $\text{nSubP}(1) \times \text{nSubP}(2) \times \text{nVars} \times \text{nEsem} \times \text{nPatch}(1) \times \text{nPatch}(2)$ . Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.

- `.x` is  $\text{nSubP}(1) \times 1 \times 1 \times \text{lnPatch}(1) \times 1$  array of the spatial locations  $x_i$  of the microscale  $(i, j)$ -grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales??
- `.y` is similarly  $1 \times \text{nSubP}(2) \times 1 \times 1 \times 1 \times \text{nPatch}(2)$  array of the spatial locations  $y_j$  of the microscale  $(i, j)$ -grid points in every patch. Currently it *must* be an equi-spaced lattice on both macro- and micro-scales.
- `.Xlim` ??

## Output

- `dudt` is a vector/array of of time derivatives, but with patch edge-values set to zero. It is of total length  $2 \cdot \text{prod}(\text{nPatch}) + \text{prod}(\text{nSubP}) \cdot \text{nVars} \cdot \text{nEnsem} \cdot \text{prod}(\text{nPatch})$  and the same dimensions as `u`.

Extract the  $2 \cdot \text{prod}(\text{nPatch})$  displacement values from the start of the vectors of evolving variables. Reshape the rest as the fields `u` in a 6D-array, and sets the edge values from macroscale interpolation of centre-patch values. ?? describes `patchEdgeInt2()`.

```

104 Nx = size(patches.x,5);
105 Ny = size(patches.y,6);
106 nM = Nx*Ny;
107 M.Dx = reshape(u( 1:nM ),[1 1 1 1 Nx Ny]);
108 M.Dy = reshape(u(nM+1:2*nM),[1 1 1 1 Nx Ny]);
109 u = patchEdgeInt2(u(2*nM+1:end),patches);

```

**Moving mesh velocity** Developing from standard moving meshes for PDEs (Budd et al. 2009, Huang & Russell 2010, e.g.), we follow Maclean, Bunder, Kevrekidis & Roberts (2021), and generalise ad hoc to 2D?? There exists a set of macro-scale mesh points  $(X_{IJ}(t), Y_{IJ}(t)) := (X_{IJ}^0 + Dx_{IJ}(t), Y_{IJ}^0 + Dy_{IJ}(t))$  (at the centre) of each patch with associated field values, say  $U_{IJ}(t) := \overline{u_{ijIJ}(t)}$ . And remove the two microscale dimensions from the front of the arrays, so they are 4D arrays.

```

128 X = shiftdim( mean(patches.x,1)+M.Dx ,2);
129 Y = shiftdim( mean(patches.y,2)+M.Dy ,2);
130 U = shiftdim( mean(mean(u,1,'omitnan'),2,'omitnan') ,2);
131 %Uz=squeeze(U)

```

Then for every patch  $(I, J)$  we set  $H_{IJ}^{pq} :=$  the  $q$ th spatial component of the step to the next patch in the  $p$ th index direction, for periodic patch indices  $(I, J)$ ,

```

140 I=1:Nx; Ip=[2:Nx 1]; Im=[Nx 1:Nx-1];
141 J=1:Ny; Jp=[2:Ny 1]; Jm=[Ny 1:Ny-1];
142 Hix = X(:, :, Ip, J)-X(:, :, I, J);
143 Hiy = Y(:, :, Ip, J)-Y(:, :, I, J);
144 Hjx = X(:, :, I, Jp)-X(:, :, I, J);
145 Hjy = Y(:, :, I, Jp)-Y(:, :, I, J);
146 Hix(:, :, Nx, :) = Hix(:, :, Nx, :)+diff(patches.Xlim(1:2));
147 Hjy(:, :, :, Ny) = Hjy(:, :, :, Ny)+diff(patches.Xlim(3:4));

```

we discretise a moving mesh PDE for node locations  $(X_{IJ}, Y_{IJ})$  with field values  $U_{IJ}$  via the second derivatives estimates ??

$$U_j'' := \frac{2}{H_j + H_{j-1}} \left[ \frac{U_{j+1} - U_j}{H_j} - \frac{U_j - U_{j-1}}{H_{j-1}} \right]. \quad (2a)$$

First, compute first derivatives at  $(I + \frac{1}{2}, J)$  and  $(I, J + \frac{1}{2})$  respectively.

```

162 Ux = (U(:, :, Ip, J)-U(:, :, I, J))./Hix(:, :, I, J); %ux=squeeze(Ux)
163 Uy = (U(:, :, I, Jp)-U(:, :, I, J))./Hjy(:, :, I, J); %uy=squeeze(Uy)

```

Second, compute second derivative matrix, without assuming symmetry because the derivatives in space are not quite the same as the derivatives in indices. The mixed derivatives are at  $(I + \frac{1}{2}, J + \frac{1}{2})$ , so average to get at patch locations.

```

173 Uxx = ( Ux(:, :, I, J)-Ux(:, :, Im, J) )*2./(Hix(:, :, I, J)+Hix(:, :, Im, J));
174 Uyy = ( Uy(:, :, I, J)-Uy(:, :, I, Jm) )*2./(Hjy(:, :, I, J)+Hjy(:, :, I, Jm));
175 Uyx = ( Uy(:, :, Ip, J)-Uy(:, :, I, J) )./Hix(:, :, I, J);
176 Uxy = ( Ux(:, :, I, Jp)-Ux(:, :, I, J) )./Hjy(:, :, I, J);
177 Uyx = (Uyx(:, :, I, J)+Uyx(:, :, Im, J)+Uyx(:, :, I, Jm)+Uyx(:, :, Im, Jm))/4;
178 Uxy = (Uxy(:, :, I, J)+Uxy(:, :, Im, J)+Uxy(:, :, I, Jm)+Uxy(:, :, Im, Jm))/4;
179 %uxx=squeeze(Uxx), uyy=squeeze(Uyy), uxy=squeeze(Uxy), uyx=squeeze(Uyx),

```

And compute its norm over all variables and ensembles (arbitrarily?? chose the mean square norm here, using `abs.^2` as they may be complex), shifting the variable and ensemble dimensions out of the result to give 2D array of values, one for each patch (use `shiftdim` rather than `squeeze` as users may invoke a 1D array of 2D patches, as in channel dispersion).

```

191 U2 = shiftdim( mean(mean( ...
192     abs(Uxx).^2+abs(Uyy).^2+abs(Uxy).^2+abs(Uyx).^2 ...
193     ,1),2) ,2);
194 Hix = shiftdim(Hix,2);  Hiy = shiftdim(Hiy,2);
195 Hjx = shiftdim(Hjx,2);  Hjy = shiftdim(Hjy,2);

```

Having squeezed out all microscale information, the global moderating coefficient in 1D??

$$\alpha := \max \left\{ 1, \left[ \frac{1}{b-a} \sum_j H_{j-1} \frac{1}{2} \left( U_j''^{2/3} + U_{j-1}''^{2/3} \right) \right]^3 \right\} \quad (2b)$$

generalises to an integral over *approximate* parallelograms in 2D?? (area approximately?? determined by cross-product). Rather than  $\max(1, \cdot)$  surely better to use something smooth like  $\sqrt{1 + \cdot^2}$ ??

```

212 U23 = U2.^(1/3);
213 alpha = sum(sum( ...
214     abs( Hix(Im,Jm).*Hjy(Im,Jm)-Hiy(Im,Jm).*Hjx(Im,Jm) ) ...
215     .*( U23(I,J)+U23(Im,J)+U23(I,Jm)+U23(Im,Jm) )/4 ...
216     ))/diff(patches.Xlim(1:2))/diff(patches.Xlim(3:4));
217 alpha = sqrt(1+alpha^6);

```

Then the importance function at each patch is the 2D array

$$\rho_j := \left( 1 + \frac{1}{\alpha} U_j''^2 \right)^{1/3}, \quad (2c)$$

```

227 rho = ( 1+U2/alpha ).^(1/3);

```

For every patch, we move all micro-grid points according to the following velocity of the notional macro-scale node of that patch: (Since we differentiate the importance function, maybe best to compute it above at half-grid points of the patches—aka a staggered scheme??)

$$V_j := \frac{dX_j}{dt} = \frac{(N-1)^2}{2\rho_j\tau} [(\rho_{j+1} + \rho_j)H_j - (\rho_j + \rho_{j-1})H_{j-1}]. \quad (2d)$$

Is the  $N_x$  and  $N_y$  correct here?? And are the derivatives appropriate since these here are scaled index derivatives, not actually spatial derivatives??

```

246 M.Vx = nan+M.Dx;  M.Vy = nan+M.Dy;  % allocate storage
247 M.Vx(:) = ( (rho(Ip,J)+rho(I,J)).*Hix(I,J) ...
248             -(rho(Im,J)+rho(I,J)).*Hix(Im,J) ) ...
249             ./rho(I,J) *(Nx^2/2/patches.mmTime);
250 M.Vy(:) = ( (rho(I,Jp)+rho(I,J)).*Hjy(I,J) ...
251             -(rho(I,Jm)+rho(I,J)).*Hjy(I,Jm) ) ...
252             ./rho(I,J) *(Ny^2/2/patches.mmTime);
253 %Vx=squeeze(M.Vx), Vy=squeeze(M.Vy), return

```

**Evaluate system differential equation** Ask the user function for the time derivatives computed in the array, overwrite its edge values with the dummy value of zero (as `ode15s` chokes on NaNs), then return to the user/integrator as same sized array as input.

```

268 dudt = patches.fun(t,u,M,patches);
269 dudt([1 end],:,:,:) = 0;
270 dudt(:, [1 end],:,:) = 0;
271 dudt=[M.Vx(:); M.Vy(:); dudt(:)];

```

Fin.

## References

- Budd, C. J., Huang, W. & Russell, R. D. (2009), ‘Adaptivity with moving grids’, *Acta Numerica* **18**, 111–241.
- Huang, W. & Russell, R. D. (2010), *Adaptive moving mesh methods*, Vol. 174, Springer Science & Business Media.
- Maclean, J., Bunder, J. E., Kevrekidis, I. G. & Roberts, A. J. (2021), Adaptively detect and accurately resolve macro-scale shocks in an efficient equation-free multiscale simulation, Technical report, University of Adelaide.
- Maclean, J., Bunder, J. E. & Roberts, A. J. (2021), ‘A toolbox of equation-free functions in matlab/octave for efficient system level simulation’, *Numerical Algorithms* **87**, 1729–1748.
- Roberts, A. J., MacKenzie, T. & Bunder, J. (2014), ‘A dynamical systems approach to simulating macroscale spatial dynamics in multiple dimensions’, *J. Engineering Mathematics* **86**(1), 175–207.  
<http://arxiv.org/abs/1103.1187>