First draft Moving Mesh documentation

AJR

August 24, 2021

Contents

1 `mm1dBurgersExample`: example of moving patches for Burgers' PDE

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.

```
clear all
global patches
patches = configPatches1(@mmBurgersPDE,[0 1], nan, 15, 0, 0.2, 5 ...
    ,'EdgyInt',true);
patches.mmTime=1;
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, $\tau \lesssim \pi^2H^2/\epsilon \approx 2$.

  (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.

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

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

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

Plots Choose whether to save some plots, or not.

```
global OurCf2eps
OurCf2eps = false;
```

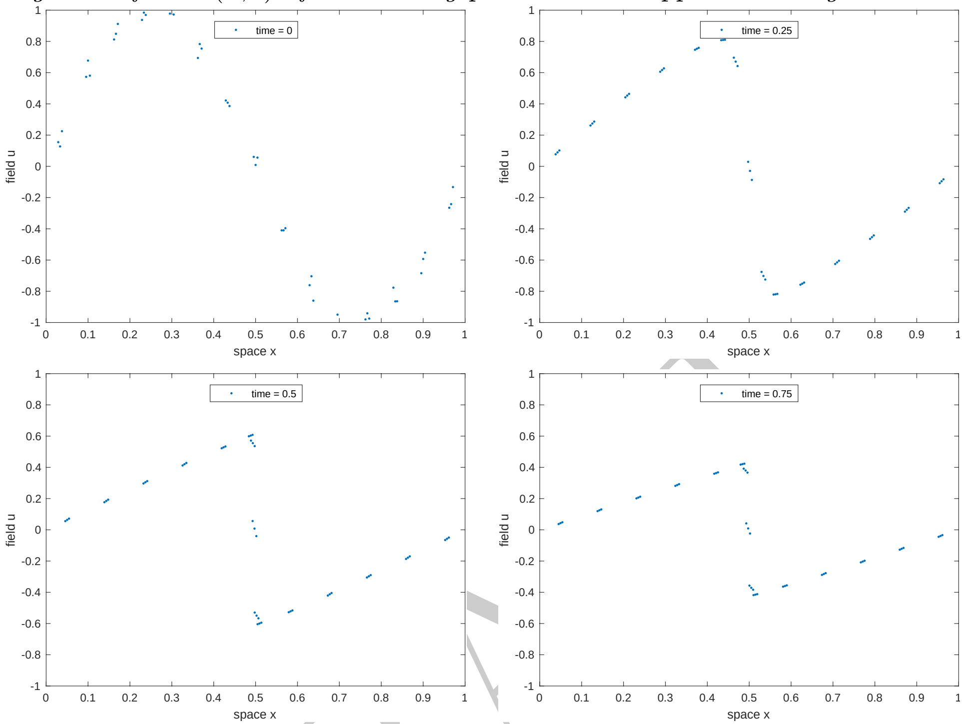
Plot the movement of the mesh, the centre of each patch, as a function of time: spatial domain horizontal, and time vertical.

```
figure(1),clf
Ds=us(:,1:N);
Xs=shiftdim(mean(patches.x),2);
plot(Xs+Ds,ts), ylabel('time t'),xlabel('space x')
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.

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

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




```
        axis([0 1 -1 1])
    else set(hpts,'XData',xs(:),'YData',us(:,i));
    end
    legend(['time = ' num2str(ts(i),2)],'Location','north')
    if rem(i,31)==1, ifOurCf2eps([mfilename num2str(i)]), end
    pause(0.04)
end
%%
```

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

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

```
f0 = mmPatchSys1(0,u0);
normf0=norm(f0)
```

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

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

Construct Jacobian with numerical differentiation.

```
deltau=1e-7;
Jac=nan(nJac);
for j=1:nJac
    uj=u0; uj(i(j))=uj(i(j))+deltau;
    fj = mmPatchSys1(0,uj);
    Jac(:,j)=(fj(i)-f0(i))/deltau;
end
```

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

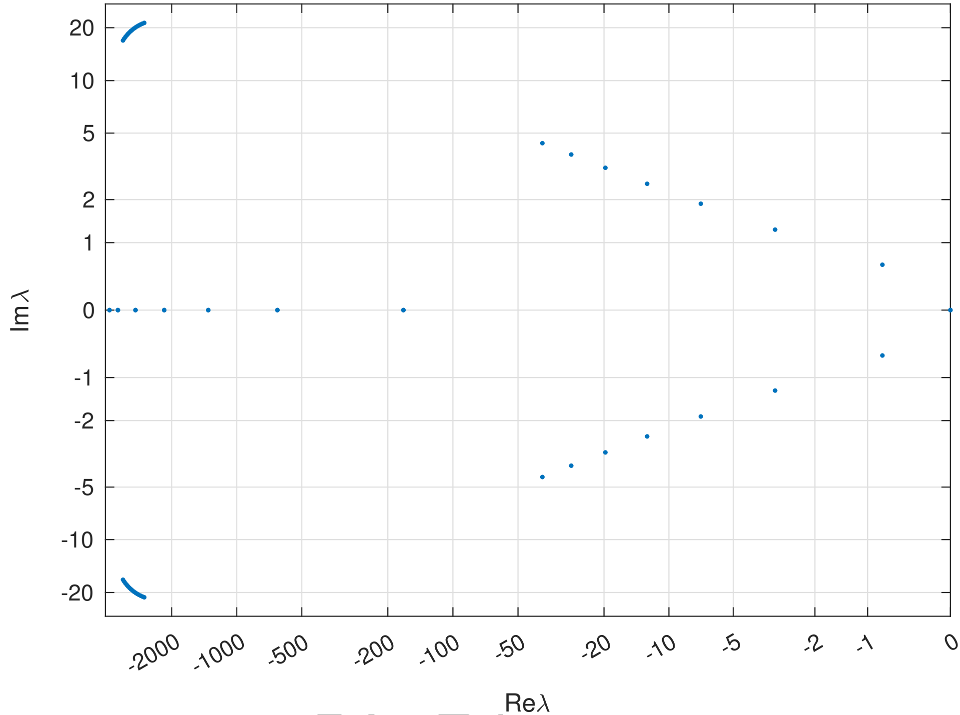
```
eval=-sort(-eig(Jac))
figure(3),clf
plot(asinh(real(eval)),asinh(imag(eval)),'.')
xlabel('Re\lambda'), ylabel('Im\lambda')
ticks=[1;2;5]*10.^(0:4);
ticks=sort([0;ticks(:);-ticks(:)]);
set(gca,'Xtick',asinh(ticks) ...
    ,'XtickLabel',cellstr(num2str(ticks,4)) ...
    ,'XTickLabelRotation',30)
set(gca,'Ytick',asinh(ticks) ...
    ,'YtickLabel',cellstr(num2str(ticks,4)))
grid
ifOurCf2eps([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.*




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

Generic input/output variables

- `t` (scalar) current time—not used here as the PDE has no explicit time dependence (autonomous).
- `u` ($n \times 1 \times 1 \times N$) field values on the patches of microscale lattice.
- `M` a struct of the following components.
  - `V` ($1 \times 1 \times 1 \times N$) moving velocity of the jth patch.
  - `D` ($1 \times 1 \times 1 \times N$) displacement of the jth 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`).

```
  u=squeeze(u);       % omit singleton dimensions
  V=shiftdim(M.V,2); % omit two singleton dimens
```

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.

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

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

Input

- `u` is a vector of length `nPatch+nSubP·nVars·nEnsem·nPatch` where there are `nVars·nEnsem` field values at each of the points in the `nSubP` × `nPatch` grid, and because of the moving mesh there are an additional `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 jth 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 `nSubP` × `nVars` × `nEnsem` × `nPatch`. Time derivatives should be computed into the same sized array, then herein the patch edge values are overwritten by zeros.
  - `.x` is `nSubP` × 1 × 1 × `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 `nPatch` + `nSubP` · `nVars` · `nEnsem` · `nPatch`.

Extract the `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()`.

```
N = size(patches.x,4);
M.D = reshape(u(1:N),[1 1 1 N]);
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 $U_j(t) := \overline{u_{ij}(t)}$.

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

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

```
j=1:N; jp=[2:N 1]; jm=[N 1:N-1];
H = X(:,:,:,jp)-X(:,:,:,j);
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

$$U''_j := \frac{2}{H_j + H_{j-1}} \left[\frac{U_{j+1} - U_j}{H_j} - \frac{U_j - U_{j-1}}{H_{j-1}} \right], \tag{1a}$$

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

```
U2 = ( (U(:,:,:,jp)-U(:,:,:,j))./H(:,:,:,j) ...
      -(U(:,:,:,j)-U(:,:,:,jm))./H(:,:,:,jm) ...
     )*2./(H(:,:,:,j)+H(:,:,:,jm));
U2 = squeeze( mean(mean( abs(U2).^2 ,2),3) );
H = squeeze(H);
```

Having squeezed out all microscale information, the coefficient

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

Rather than $\max(1,\cdot)$ surely better to use something smooth like $\tanh(\cdot)$??

```
alpha = sum( H(jm).*( U2(j).^(1/3)+U2(jm).^(1/3) ))/2/sum(H);
alpha = max(1,alpha^3);
```

Then the importance function

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

```
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:

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

```
M.V = nan+M.D; % allocate storage
M.V(:) = ( (rho(jp)+rho(j)).*H(j) -(rho(j)+rho(jm)).*H(jm) ) ...
    ./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.

```
dudt=patches.fun(t,u,M,patches);
dudt([1 end],:,:,:) = 0;
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 6×4-periodic domain, with 9×7 patches, spectral interpolation (0) couples the patches, each patch of half-size ratio 0.4 (relatively large for visualisation), and with 5×5 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.

```
clear all
global patches
nxy=5
patches = configPatches2(@mmNonDiffPDE,[-3 3 -2 2], nan ...
    , [9 7], 0, 0.4, nxy ,'EdgyInt',true);
patches.mmTime=1;
patches.Xlim=[-3 3 -2 2];
```

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

If we use more patches, then the algorithm goes berserk after some time??

Decide the moving mesh time parameter

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

```
u0 = exp(-patches.x.^2-patches.y.^2);
u0 = u0.*(0.9+0.1*rand(size(u0)));
Nx = size(patches.x,5)
Ny = size(patches.y,6)
Npts = Nx*Ny;
D0 = zeros(2*Npts,1);
```

Integrate in time to $t = 4??$ 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.

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

Plots Choose whether to save some plots, or not.

```
global OurCf2eps
OurCf2eps = false;
```

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

```
nTime=length(ts);
Ds=reshape(us(:,1:2*Npts).',Nx,Ny,2,nTime);
us=reshape(us(:,2*Npts+1:end).',nxy,nxy,Nx,Ny,nTime);
Us=shiftdim( mean(mean(us,1),2) ,2);
%% section marker for plot execution
figure(1),clf, colormap(0.8*hsv)
Xs=shiftdim(mean(patches.x),4);
Ys=shiftdim(mean(patches.y),4);
for k=1:nTime
  Xk=Xs+Ds(:,:,1,k);
  Yk=Ys+Ds(:,:,2,k);
```

```
  if k==1,
    hand=mesh(Xk,Yk,Us(:,:,k));
    ylabel('space y'),xlabel('space x'),zlabel('mean field U')
    axis([patches.Xlim 0 1]), caxis([0 1])
    colorbar
    if 0, view(0,90) % vertical view
    else  view(-25,60) % 3D perspective
    end
  else
    set(hand,'XData',Xk,'YData',Yk ...
       ,'ZData',Us(:,:,k),'CData',Us(:,:,k))
  end
  legend(['time =' num2str(ts(k),4)],'Location','north')
  pause(0.05)
end
```

Fin.

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 $\dot u_{ijkl} = \cdots + \frac{1}{\delta x^2}(u^3_{i+1,j,k,l} - 2u^3_{i,j,k,l} + u^3_{i-1,j,k,l}) + \frac{1}{\delta y^2}(u^3_{i,j+1,k,l} - 2u^3_{i,j,k,l} + u^3_{i,j-1,k,l})$.

```
function ut = mmNonDiffPDE(t,u,M,patches)
  if nargin<3, global patches, end
  u = squeeze(u); % reduce to 4D
  Vx = shiftdim(M.Vx,2); % omit two singleton dimens
  Vy = shiftdim(M.Vy,2); % omit two singleton dimens
  dx = diff(patches.x(1:2));  % micrgrid spacing
  dy = diff(patches.y(1:2));
  i = 2:size(u,1)-1;  j = 2:size(u,2)-1;  % interior patch points
  ut = nan+u;  % preallocate output array
  ut(i,j,:,:) = ...
      +Vx.*(u(i+1,j,:,:)-u(i-1,j,:,:))/(2*dx) ...
      +Vy.*(u(i,j+1,:,:)-u(i,j-1,:,:))/(2*dy) ...
      +diff(u(:,j,:,:).^3,2,1)/dx^2 ...
      +diff(u(i,:,:,:).^3,2,2)/dy^2 ;
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`).

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

Input

- `u` is a vector of length $2 \cdot$ `prod(nPatch)` $+$ `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` where there are `nVars` $\cdot$ `nEnsem` field values at each of the points in the `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nPatch(1)` $\times$ `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 $(M.Vx_I, M.Vy_J)$ 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 `nSubP(1)` $\times$ `nSubP(2)` $\times$ `nVars` $\times$ `nEsem` $\times$ `nPatch(1)` $\times$ `nPatch(2)`. Time derivatives must be computed into the same sized array, although herein the patch edge-values are overwritten by zeros.
  - `.x` is `nSubP(1)` $\times 1 \times 1 \times$ `1nPatch(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 $1 \times$ `nSubP(2)` $\times 1 \times 1 \times 1 \times$ `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 $2 \cdot$ `prod(nPatch)` $+$ `prod(nSubP)` $\cdot$ `nVars` $\cdot$ `nEnsem` $\cdot$ `prod(nPatch)` and the same dimensions as `u`.

Extract the $2 \cdot$ `prod(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()`.

```
Nx = size(patches.x,5);
Ny = size(patches.y,6);
nM = Nx*Ny;
M.Dx = reshape(u(   1:nM  ),[1 1 1 1 Nx Ny]);
M.Dy = reshape(u(nM+1:2*nM),[1 1 1 1 Nx Ny]);
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 $\big(X_{IJ}(t), Y_{IJ}(t)\big) := \big(X^0_{IJ} + Dx_{IJ}(t), Y^0_{IJ} + Dy_{IJ}(t)\big)$ (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.

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

Then for every patch (I, J) we set H^{pq}_{IJ} := the qth spatial component of the step to the next patch in the pth index direction, for periodic patch indices (I, J),

```
I=1:Nx; Ip=[2:Nx 1]; Im=[Nx 1:Nx-1];
J=1:Ny; Jp=[2:Ny 1]; Jm=[Ny 1:Ny-1];
Hix = X(:,:,Ip,J)-X(:,:,I,J);
Hiy = Y(:,:,Ip,J)-Y(:,:,I,J);
Hjx = X(:,:,I,Jp)-X(:,:,I,J);
Hjy = Y(:,:,I,Jp)-Y(:,:,I,J);
Hix(:,:,Nx,:) = Hix(:,:,Nx,:)+diff(patches.Xlim(1:2));
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 ??

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

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

```
Ux = (U(:,:,Ip,J)-U(:,:,I,J))./Hix(:,:,I,J);
Uy = (U(:,:,I,Jp)-U(:,:,I,J))./Hjy(:,:,I,J);
```

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.

```
Uxx = ( Ux(:,:,I,J)-Ux(:,:,Im,J) )*2./(Hix(:,:,I,J)+Hix(:,:,Im,J));
Uyy = ( Uy(:,:,I,J)-Uy(:,:,I,Jm) )*2./(Hjy(:,:,I,J)+Hjy(:,:,I,Jm));
Uyx = ( Uy(:,:,Ip,J)-Uy(:,:,I,J) )./Hix(:,:,I,J);
Uxy = ( Ux(:,:,I,Jp)-Ux(:,:,I,J) )./Hjy(:,:,I,J);
Uyx = (Uyx(:,:,I,J)+Uyx(:,:,Im,J)+Uyx(:,:,I,Jm)+Uyx(:,:,Im,Jm))/4;
Uxy = (Uxy(:,:,I,J)+Uxy(:,:,Im,J)+Uxy(:,:,I,Jm)+Uxy(:,:,Im,Jm))/4;
```

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).

```
U2 = shiftdim( mean(mean( ...
         abs(Uxx).^2+abs(Uyy).^2+abs(Uxy).^2+abs(Uyx).^2 ...
     ,1),2) ,2);
Hix = shiftdim(Hix,2); Hiy = shiftdim(Hiy,2);
Hjx = shiftdim(Hjx,2); Hjy = shiftdim(Hjy,2);
```

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

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

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 $\tanh(\cdot)$??

```
U23 = U2.^(1/3);
alpha = sum(sum( ...
    abs( Hix(Im,Jm).*Hjy(Im,Jm)-Hiy(Im,Jm).*Hjx(Im,Jm) ) ...
    .*( U23(I,J)+U23(Im,J)+U23(I,Jm)+U23(Im,Jm) )/4 ...
    ))/diff(patches.Xlim(1:2))/diff(patches.Xlim(3:4));
alpha = tanh(alpha^3);
```

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

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

```
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??)

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

Is the `Nx` and `Ny` correct here?? And are the derivatives appropriate since these here are scaled index derivatives, not actually spatial derivatives??

```
M.Vx = nan+M.Dx;  M.Vy = nan+M.Dy;  % allocate storage
M.Vx(:) = ( (rho(Ip,J)+rho(I,J)).*Hix(I,J) ...
           -(rho(Im,J)+rho(I,J)).*Hix(Im,J) ) ...
          ./rho(I,J) *(Nx^2/2/patches.mmTime);
M.Vy(:) = ( (rho(I,Jp)+rho(I,J)).*Hjy(I,J) ...
           -(rho(I,Jm)+rho(I,J)).*Hjy(I,Jm) ) ...
          ./rho(I,J) *(Ny^2/2/patches.mmTime);
%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.

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