LilyPad base class reference

June 9, 2016

Abstract

.

1 Used Postprocessing objects

- `PVector` - Cartesian vector with x and y components

2 OrthoNormal

2.1 Description

Holds information describing a line segment.

2.2 Internal fields

- `<float> l` - length of the line segment
- `<float> nx` - normal vector x-component
- `<float> ny` - normal vector y-component
- `<float> tx` - x-tangent - unit vector between end points of the line segment
- `<float> ty` - y-tangent
- `<float> off` - normal offset - distance from the origin to the plane defined by this line segment
- `<float> t1` - distance of the end points projected onto the tangent
- `<float> t2`
- `<PVector> cen` - centre of the line segment in the global coordinate system

2.3 Methods

2.3.1 Constructor

Accepts two points in space and computes all the properties of the line segment they define.

```
OrthoNormal(PVector x1, PVector x2 )
{
  // length of the line segment
  l = PVector.sub(x1,x2).mag();
  tx = (x2.x-x1.x)/l;        // x tangent
  ty = (x2.y-x1.y)/l;        // y tangent
  t1 = x1.x*tx+x1.y*ty;      // tangent location of point 1
  t2 = x2.x*tx+x2.y*ty;      // tangent location of point 2
  nx = -ty;
  ny = tx;             // normal vector
  off = x1.x*nx+x1.y*ny;     // normal offset
  cen = PVector.add(x1,x2); // centriod
  cen.div(2.);
}
```

2.3.2 Distance

Computes the distance between the line described by this instance of OrthoNormal object and a gieven point. If no optional parameter is passed then a wrapper method is called which passes `projected=true`. This means that the function computes the normal distance from the point described by (x,y) by calling a dot product with the line nromal. If `projected` is set to false, first the normal distance is computed as previously. Then the distance between the intersection of normal direction of the line segment passing through the point of interest and the tangential direction of the line segment is computed. The return value is equal to the one computed using projected normal if the intersection lies on the line segment or the sum of the normal and tangential distances if the intersection point falls outside of the line segment.

```
float distance( float x, float y, Boolean projected)
{
  float d = x*nx+y*ny-off; // normal distance to line
  if(projected) return d;  // |distance|_n (signed, fastest)

  float d1 = x*tx+y*ty-t1; // tangent dis to start
  float d2 = x*tx+y*ty-t2; // tangent dis to end

  //    return sqrt(sq(d)+sq(max(0,-d1))+sq(max(0,d2))); // |distance|_2
  return abs(d)+max(0,-d1)+max(0,d2);              // |distance|_1 (faster)
}
```

2.3.3 tanCoord

Returns the distance from the point of interest, (x,y), to the start of the line segment along its tangential direction.

```
float tanCoord( float x, float y )
{
  return min( max( (x*tx+y*ty-t1)/l, 0), 1 );
}
```

3 Scale

3.1 Description

Helps with organising display of data, allows interpolation between the actual mesh and the pixelated grid. In the CFD each Cartesian axis has its own scale object.

3.2 Internal fields

- `<float> inS` - start of the inner scale
- `<float> inE` - end of the inner scale
- `<float> outS` - same as above but for the outer scale
- `<float> outE`
- `<float> r` - ratio between the units of the outer and inner scales

3.3 Methods

3.3.1 Constructor

Assembles the scale from components, explicitly assinging values to each of the fields.

```
Scale( float inS, float inE, float outS, float outE )
{
  this.inS  = inS;
  this.inE  = inE;
  this.outS = outS;
  this.outE = outE;
  r = (outE-outS)/(inE-inS);
}
```

3.3.2 OutB

Converts from the inner to outer scale but making sure the returned value is bounded between the start and end values of the outer scale.

```
float outB( float in ){ return out(min(max(in,inS),inE));}
```

3.3.3 Out

Converts the given coordinate in the inner scale into its corresponding value in the outer scale.

```
float out( float in ){ return (in-inS)*r+outS;}
```

3.3.4 In

Converts a given outer scale value into the inner scale.

```
float in( float out ){ return (out-outS)/r+inS;}
```

4 Window

4.1 Description

Helps with organising display of data, allows interpolation between the actual mesh and the pixelated grid.

4.2 Internal fields

- `<Scale> x` - scale of the x axis
- `<Scale> y` - scale of the y axis
- `<int> x0` - beginning of the outer (display) scale in pixels for the x-axis
- `<int> y0` - beginning of the outer scale for the y-axis in pixels
- `<int> dx` - dimension of the x-axis in pixels
- `<int> dy` - dimension of the y-axis in pixels

4.3 Methods

4.3.1 Constructor - from components

Assembles the scale from components, explicitly assinging values to each of the fields.

```
Window( float n0, float m0, float dn, float dm, int x0, int y0, int dx, int dy)
{
  // create the scale objects for both axes allowing interpolation onto the
  // display to be performed; inner scale is the CFD grid
  x = new Scale(n0,n0+dn,x0,x0+dx);
  y = new Scale(m0,m0+dm,y0,y0+dy);

  // assign the size and beginning of the display area
  this.x0 = x0;
  this.y0 = y0;
  this.dx = dx;
  this.dy = dy;
}
```

4.3.2 Constructor - basic

Called by the basic LilyPad example. Only accepts the number of cells in both directions and starts the inner grid from index 1. Read the size of the window using global variables width and height, assigning both display axes to start at 0.

```
Window( int n, int m)
{
  this( 1, 1, n-2, m-2, 0, 0, width, height );
}
```

4.3.3 Inside

Checks if the passed point fits within the limits of the display window.

```
boolean inside( int x, int y )
{
  return( x>=x0 && x<=x0+dx && y>=y0 && y<=y0+dy );
}
```

4.3.4 ix/iy

Converts from the cell index to pixel coordinate.

```
float ix(int i){ return x.in((float)i);}
float iy(int i){ return y.in((float)i);}
```

4.3.5 idx/idy

Inverse conversion, used for something to do with mouse coordinates.

```
float idx(int i){ return i/x.r;}
float idy(int i){ return i/y.r;}
```

4.3.6 px/py

Convert a pixel location to a cell index.

```
int px(float i){ return (int)(x.out(i));}
int py(float i){ return (int)(y.out(i));}
```

4.3.7 pdx/pdy

Inverse transfrom in the pixel axis.

```
int pdx(float i){ return (int)(x.r*i);}
int pdy(float i){ return (int)(y.r*i);}
```

5 Field

5.1 Description

Holds the information necessary to manipulate a scalar field, such as keeping its values, advecting given a velocity field, taking gradients or Laplacians, interpolating, etc.

5.2 Internal fields

- `<float[][]> a` - array of internal field values, shape (n,m)
- `<int> n, m` - number of grid points in x- and y-directions
- `<int> btype` - defines where the variable is defined in the segregated storage scheme
  - 0 - cell centre (e.g. pressure)

  - 1 - negative x-face (e.g. x-velocity)
  - 2 - negative y-face (e.g. y-velocity)

- `<float> bval` - value of the boundary condition (only used in certain cases)
- `<bool> gradientExit` - flag specifying whether gradient outlet BC is used; important only if `btype` is 1 or 2, namely a vector field component is being considered

5.3 Methods

5.3.1 Constructor - from components

Allocate memory to the field value array and set it equal to the specified boundary value, `bval`.

```
Field( int n, int m, int btype, float bval )
{
  this.n = n;
  this.m = m;
  this.a = new float[n][m];
  this.btype = btype;
  this.bval = bval;
  // sets the internal field values to bval
  this.eq(bval);
}
```

5.3.2 Laplacian

Compute second spatial derivative of the field,

$$\nabla^2\phi = \frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2}, \tag{1}$$

using a 2nd order central scheme.

```
Field laplacian ()
{
  // create an empty field d filled with zeros
  Field d = new Field( n, m );

  // loop over all INTERNAL cells (mind the end stencils)
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // use a 2nd order linear scheme in x- and y-directions
      // phi''(x) = [phi(x-delta_x) - 2 phi(x) + phi(x+delta_x)] / delta_x^2
      // recall that the code is dimensionless with
      // the grid size being the length-scale
      // this means we skip the delta_x and delta_y
      // in the denominator
      d.a[i][j] = -4*a[i][j] + a[i+1][j]
        + a[i-1][j] + a[i][j+1] + a[i][j-1];
    }
  }
  // leave the boundary values equal to zero
  return d;
}
```

5.3.3 Gradient

Compute the first spatial derivative of the field,

$$\nabla\phi = \frac{\partial\phi}{\partial x}\hat{\mathbf{i}} + \frac{\partial\phi}{\partial y}\hat{\mathbf{j}}, \tag{2}$$

using a 2nd order central scheme.

```
VectorField gradient()
{
  // this will hold the grad(a) values, initialise with zeros
  VectorField g = new VectorField(n,m,0,0);

  // loop over all internal faces
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // compute the x and y derivatives using a
      // 2nd order backward scheme (recall that gradients are stored at face centres so
      // g.x(x-delta_x/2,y) = (phi(x,y)-phi(x-delta_x,y)) / delta_x
      // phi'(x) = [phi(x) - phi(x-delta_x)] / delta_x
      g.x.a[i][j] = a[i][j]-a[i-1][j];
      g.y.a[i][j] = a[i][j]-a[i][j-1];
    }
  }

  // call the vectorField method to set the boundary values to zero in both directions (
      gradientExit=false)
  // this calls the Field::setBC() function on both x- and
  // y-components of the gradient function; x-dir is given
  // btype 1 and y-dir btype 2 - this causes the field values to be stored at the faces
  // as per staggered mesh formulation for a vector field
  g.setBC(); // issues?
  return g;
}
```

5.3.4 Advect

Given a velocity field at this time step, `(u,v)`, and at the previous time step, `(u0,v0)`, convect the scalar field in space. This uses the basic form of the scalar transport equation,

$$\frac{\mathrm{D}\phi}{\mathrm{D}t} = 0, \tag{3}$$

i.e. total derivative equals zero, assuming there are no source terms on the RHS of the equation. To be more accurate, the function uses two consecutive time step values with a 2nd order Runge-Kutta method and a quadratic interpolation routine in space.

```
void advect(float step, Field u, Field v, Field u0, Field v0)
{
  /* advect the field with the u, v velocity fields.
  Use a first order lagrangian method:
  Da/Dt = 0
  which translates into:
  a(t=dt,x,y) = a(t=0, x-dt*u(x,y), y-dt*v(x,y))

  The example code shows how diffusive this is
  EDIT: by using an RK2 step to find x0 and using
  a quadratic interpolation, this method is second
  order and nondiffuse.
  EDIT2: by treating the old and new velocities
  seperately, the method is second order in time.*/

  // create a copy of this field, keeping the current values
  Field a0 = new Field(this);

  // loop over all internal cells
  for( int i=1; i<n-1; i++)
  {
    for( int j=1; j<m-1; j++)
    {
      // select the interpolation point to be the +ve face of cell 0,
      // iterating up to -ve face of the last cell - i.e. all internal faces
      float x = i;
      float y = j;
      // correct for boundary values if necessary
      // Note: linear and quadratic interpolation methods will modify x and y by +0.5
      // if the field is stored at the -ve face centres because we want to convert the x-
      // and y-location into indices in the matrix. The interpolation method is generic
          and
      // so also makes sure the point at which interpolation is performed is valid,
          bounded inside
      // the domain and so on. If btype>0 the variable is stored at face centre, and hence
           0.5\delta
      // before/below the (x,y) of the (i,j) cell
      if(btype==1) x -= 0.5;
      if(btype==2) y -= 0.5;
      // get the velocities at this time step
      float ax = -step*u.linear( x, y );
      float ay = -step*v.linear( x, y );
      // get the velocities at the previous time step
      // account for the fact that the interpolation point would have
      // "moved" back then as well
      float bx = -step*u0.linear( x+ax, y+ay );
      float by = -step*v0.linear( x+ax, y+ay );
      // assume that at the new time step field value is equal to
      // that at the previous time step at a location -dT*(u,v)
      // perform Runge-Kutta interpolation using three data points
      // to get the location in the field which would have been the current
      // value one time step ago
      // then use quadratic interpolation to get the field value there and
      // assign it to be the field value after update
      a[i][j] = a0.quadratic( x+0.5*(ax+bx), y+0.5*(ay+by) );
    }
  }
  // correct the BCs
  setBC();
}
```

5.3.5 Quadratic

Given a point of interest, (x0,y0), interpolate the field values in space to get a value at that location. The code uses

```
float quadratic( float x0, float y0)
{
  // correct (x0,y0) if the field is stored at face centres
  float x = x0, y = y0;
  if(btype==1) x += 0.5;
  if(btype==2) y += 0.5;

  // get the indices of cells we need
  int i = round(x), j = round(y);

  // make sure we won't exceed the grid dimensions
  // perform linear interpolation if we do
  if( i>n-2 || i<1 || j>m-2 || j<1 )
    return linear( x0, y0 );

  // to interpolate we need x and y expressed about
  // the i-th and j-th cell
  x -= i;
  y -= j;

  // interpolate in the x-direction at 3 y-positions
  float e = quadratic1D(x,a[i-1][j-1],a[i][j-1],a[i+1][j-1]);
  float f = quadratic1D(x,a[i-1][j  ],a[i][j  ],a[i+1][j  ]);
  float g = quadratic1D(x,a[i-1][j+1],a[i][j+1],a[i+1][j+1]);

  // use the interpolated values to interpolate in the y-direction
  return quadratic1D(y,e,f,g);
}
```

5.3.6 Quadratic 1D

Quadratic interpolation in 1D given a point of interest, `x`, and three field values at consecutive grid points. The scheme is bounded between the data points used for interpolation.

```
float quadratic1D(float x, float e, float f, float g)
{
  // compute the value
  float x2 = x*x;
  float fx = f*(1.-x2);
  fx += (g*(x2+x)+e*(x2-x))*0.5;
  // bound the scheme between the specified field values
  fx = min(fx,max(e,f,g));
  fx = max(fx,min(e,f,g));
  return fx;
}
```

5.3.7 Linear

Basic linear interpolation of the field values at a point of interest.

```
// Interpolate the field at point (x0,y0)
float linear(float x0, float y0)
{
  // take the point of interest and make it bound between
  // 0.5 and x_max-0.5 - i.e. cell centres of 0th and N-1st cell (boundary cells)
  float x  = min(max(0.5,x0), n-0.5);
  if(btype==1) x += 0.5;

  // index of the cell around which we interpolate
  // make sure we do not pick N-1st cell (the last one)
  // casting <float>x to <int> will round down to the nearest integer
  // so that x should be between i and i+1
  int i = min( (int)x, n-2 );
  // weighting factor in the x-direction
  float s = x-i;

  float y  = min(max(0.5,y0), m-1.5);
  if(btype==2) y += 0.5;

  int j = min( (int)y, m-2 );
  float t = y-j;

  // if weighting factors are zero then just pick the field value (i,j)
  if(s==0 && t==0)
  {
    return a[i][j];
  }
  // perform linear interpolation
  // first do in the y-direction at both x-locations we need
  // then take the results and interpolate in the x-direction
  else
  {
    return  s*(t*a[i+1][j+1]+(1-t)*a[i+1][j])+
      (1-s)*(t*a[i  ][j+1]+(1-t)*a[i  ][j]);
  }
}
```

5.3.8 Set BC

This function takes care of all boundary condition related operations. First apply von Neumann (normal gradient equal to zero) in both x- and y-directions by setting the first and last grid point values to be equal to those in the second and second-last cells. If the variable is stored at cell centres this is the only step required. In the case of staggered variables, such as velocity x- and y-components, values at the inlet and outlet faces are also prescribed. In a standard application with flow from left to right, these would imply zero velocities in the y-direction at the walls, a fixed value BC at the left-hand side, and an outlet BC at the RHS (`gradientExit` set to `true`). outlet BC is not your every-day zeroGradient, so what is it?

```
void setBC ()
{
  float s=0;

  // go over all y-indices
  for (int j=0 ; j<m ; j++ )
  {
    // set the 0th and n-1st (first and last) field values
    // to what they are one index into the domain
    // -> apply zero-gradient (von Neumann) BC at left and right sides
    a[0][j]   = a[1][j];
    a[n-1][j] = a[n-2][j];

    // if the field values are stored at the faces
    if(btype==1)
    {
      // if we have a zero-gradient type right-hand-side boundary
      // but fixed-value inlet then prescribe the value only there
      if(gradientExit)
      {
        // set inlet to fixed-value
        a[1][j] = bval;
        // s now holds an integral of the field values at the RHS boundary
        if(j>0 & j<m-1)
          s += a[n-1][j];
      }
      // set the field values to the constant held in the class
      // if we have a fixed-value type BC set
      // do this to cells 1st cell into the domain at the LHS and last one at the RHS
      // this ensures zero-gradient at the LHS is maintained later on at the LHS
      else
      {
        a[1][j]   = bval;
        a[n-1][j] = bval;
      }
    }
  }

  // go over all x-indices
  for (int i=0 ; i<n ; i++ )
  {
    // apply zero-gradient in the y-direction at top and bottom
    a[i][0]   = a[i][1];
    a[i][m-1] = a[i][m-2];

    // if the field values are stored at face centres set boundary values
    if(btype==2)
    {
      a[i][1]   = bval;
      a[i][m-1] = bval;
    }
  }

  // set the RHS BC
  if(gradientExit)
  {
    // get the average values of s
    s /= float(m-2);
    // remove the mean value from the RHS cell and the boundary value
    for( int j=1; j<m-1; j++ )
      a[n-1][j] += bval-s;
  }
}
```

5.3.9 Normal gradient

Compute gradient in a normal direction where `wnx` and `wny` are wall-normal x- and y-directions with their `VectorField::x` holding the actual values of the gradient. See the function `BDIM::get_wn` for more information on how these are calculated.

The evaluated expression is

$$g = \mathbf{w}_n \cdot \nabla\phi \tag{4}$$

where the gradient of the field is approximated using central scheme.

```
Field normalGrad(VectorField wnx, VectorField wny)
{
  // result field
  Field g = new Field(n,m,0,0);

  // loop over all internal cells
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // compute the wall-normal gradient
      // approximate d(a)/dx_i using 2nd order central scheme
      // [a(i+1,j)-a(i-1,j)]/(2 delta_x), where delta_x = 1.0
      // take a dot-product with the wall-normal direction
      g.a[i][j] = 0.5*( wnx.x.a[i][j] * (a[i+1][j  ]-a[i-1][j  ])
              + wny.x.a[i][j] * (a[i  ][j+1]-a[i  ][j-1]) );
    }
  }
  return g;
}
```

6 Vector field

6.1 Description

Class which makes use of the scalar field type to implement functionality relevant to a vector quantity in 2D space. It has some of its methods similar to those of a scalar field, such as assignment or algebraic operators, normal gradient computation, etc. which are omitted here fore brevity.

6.2 Internal fields

- `<Field> x, y` - x- and y-components of the vector quantity, stored as `Field` types
- `<int> n, m` - no. cells in x- and y-directions
- `<float> CF` - QUICK parameter used to fit a polynomial through three points, set to 1/6
- `<float> S` - QUICK parameter used to tune the limiter, set to 10

6.3 Methods

6.3.1 Normal gradient

This computes the gradient of the velocity components similarly to how it is done for the scalar field class.

```
// compute gradient in a normal direction
// wnx and wny are wall-normal x- and y-directions with their
// VectorField.x holding the actual values in x- and y-dirs
VectorField normalGrad(VectorField wnx, VectorField wny)
{
  VectorField g = new VectorField(n,m,0,0);
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      g.x.a[i][j] = 0.5*(wnx.x.a[i][j]*(x.a[i+1][j  ]-x.a[i-1][j  ])
                +wny.x.a[i][j]*(x.a[i  ][j+1]-x.a[i  ][j-1]));
      g.y.a[i][j] = 0.5*(wnx.y.a[i][j]*(y.a[i+1][j  ]-y.a[i-1][j  ])
                +wny.y.a[i][j]*(y.a[i  ][j+1]-y.a[i  ][j-1]));
    }
  }
  return g;
}
```

6.3.2 Divergence

What else, computes divergence of the field, in other words

$$\nabla \cdot \mathbf{U} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}. \tag{5}$$

The derivative operator is approximated using 2nd order central difference.

```
Field divergence ()
{
  // create the returned field
  Field d = new Field( n, m );
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // use 2nd order central difference to get the derivative
      // recall the vector field value (i) is at the negative face of cell (i)
      // therefore central difference at cell centre is (phi(i+1,j)-phi(i,j)) / delta_x
      d.a[i][j] = x.a[i+1][j  ]-x.a[i][j]+
            y.a[i  ][j+1]-y.a[i][j];
    }
  }
  return d;
}
```

6.3.3 Turbulent KE

Computes a field which would return the turbulent kinetic energy if the stored field was actually a velocity field. Typically, this evaluates to

$$TKE = \frac{1}{4}\left(u'^2 + v'^2\right), \tag{6}$$

where prime variables denote quantities fluctuating about the ensemble average at a given point.

```
Field ke ()
{
  Field d = new Field( n, m );
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // first, take the average velocity in x- and y-directions; note that in the stream-
          wise
      // direction need to subtract the mean to get the fluctuating component; this has
          unit value
      // at faces i and i+1, therefore we subtract 2 from their sum
      // then square each fluctuating component and add; division by 2 taken in front of
          the square
      // operator, hence turns into 0.25
      d.a[i][j] = ( sq(x.a[i+1][j  ]+x.a[i][j]-2.0)
            + sq(y.a[i  ][j+1]+y.a[i][j]    ) )*0.25;
    }
  }
  return d;
}
```

6.3.4 Vorticity

Computes a field which would return the vorticity if the stored field was actually a velocity field. In other words, the result is

$$\nabla \times \mathbf{U} = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}. \tag{7}$$

Gradients are computed using central scheme.

```
Field vorticity ()
{
  Field d = new Field( n, m );
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // take the gradients and then compute the value at the cell centre
      // Note: the flow is 2D, in 3D the result would be interpreted as the k_hat
         component of a vector field
      // gradients approximated using central difference
      // we need to first calculate the gradient at the positive and negative face
      // by using neighbouring faces; for instance, for +ve x face we take faces at (i+1,j
         -1)
      // and (i+1,j+1), subtract and divide by 2 delta_y; we repeat the same procedure at
      // (i,j-1) and (i,j+1) and take the mean of the two values; proceeding analogously
         in
      // the y-drection and taking the two divisions by 2 yields 0.25 and the following
      d.a[i][j] = 0.25*(  x.a[i  ][j-1]-x.a[i  ][j+1]+ // -du/dy
                x.a[i+1][j-1]-x.a[i+1][j+1]+ // -du/dy at face i+1
                y.a[i+1][j  ]-y.a[i-1][j  ]+ // dv/dx
                y.a[i+1][j+1]-y.a[i-1][j+1]); // dv/dx at face j+1
    }
  }
  return d;
}
```

6.3.5 Q criterion

Computes a field which would return the Q criterion if the stored field was actually a velocity field. To define it, let us first denote the rate-of-strain tensor,

$$\mathbf{S} = \frac{1}{2}\left[\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right] = \frac{1}{2}\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial v}{\partial x} \\ \frac{\partial u}{\partial y} & \frac{\partial v}{\partial y} \end{bmatrix} + \frac{1}{2}\begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix} = \frac{1}{2}\begin{bmatrix} 2\frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \\ \frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} & 2\frac{\partial v}{\partial y} \end{bmatrix}, \tag{8}$$

and vorticity tensor,

$$\mathbf{\Omega} = \frac{1}{2}\left[\nabla\mathbf{U} - (\nabla\mathbf{U})^T\right] = \frac{1}{2}\begin{bmatrix} 0 & \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \\ \frac{\partial u}{\partial y} - \frac{\partial v}{\partial x} & 0 \end{bmatrix}. \tag{9}$$

Now, the vortex identification criterion may be written as

$$Q = \frac{1}{2}\left(|\mathbf{\Omega}|^2 - |\mathbf{S}|^2\right) > 0. \tag{10}$$

should this not be du_i/dx_j du_j/dx_i but without excluding the option with i=j? also, would there be a minus between the two terms? like eq. 1 in: http://www.wseas.us/e-library/conferences/2011/Mexico/MAFLUH/MAFLUH-03.pdf

```
Field Qcrit ()
{
  Field q = new Field( n, m );
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // compute finite differences at the (i,j) face using central scheme
      float dudx = 0.5*(x.a[i+1][j  ]-x.a[i-1][j  ]);
      float dudy = 0.5*(x.a[i  ][j+1]-x.a[i  ][j-1]);
      float dvdx = 0.5*(y.a[i+1][j  ]-y.a[i-1][j  ]);
      float dvdy = 0.5*(y.a[i  ][j+1]-y.a[i  ][j-1]);
      q.a[i][j] = dudx*dvdy-dvdx*dudy;
    }
  }
  return q;
}
```

6.3.6 Diffusion

Physically, this represents energy dissipation in the fluid. By assuming incompressibility, this becomes

$$\nu\nabla^2\mathbf{U} = \frac{\partial^2 u_k}{\partial x_i x_j}\hat{\mathbf{e}}_k\delta_{ij} = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)\mathbf{i} + \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)\mathbf{j}. \tag{11}$$

Note that this function computes the Laplacian for a single component of a vector field only. This is done using a second order linear scheme to compute the 2$^{\text{nd}}$ spatial derivative around cell. In one dimension this can be written as

$$\frac{\partial^2\phi}{\partial x^2} \approx \phi(x-\Delta x) - 2\phi(x) + \phi(x+\Delta x) + O[\Delta x^2]. \tag{12}$$

```
float diffusion (Field b, int i, int j)
{
  // use second order linear scheme to compute the 2nd spatial derivative around cell i,j
  // compute d^2(phi)/d(x_i^2) values in the x- and y-directions and add
  return b.a[i+1][j] + b.a[i][j+1] - 4*b.a[i][j] + b.a[i-1][j] + b.a[i][j-1];
}
```

6.3.7 Limiter

This function accepts a value a and makes sure this does not exceed the prescribed upper and lower limits given as b and c, respectively.

```
float med(float a, float b, float c)
{
  return(max(min(a, b), min(max(a, b), c)));
}
```

6.3.8 Face value interpolation

In order to compute sum of the fluxes on each cell face, the cell-centre values need to be interpolated. The code uses Quadratic Upwind Interpolation for Convective Kinematics (QUICK) scheme in order to accomplish this. For example, for the east face of a cell P located at x this would be

$$\phi(x+\Delta x/2) \approx \begin{cases} \frac{1}{2}(\phi(x+\Delta x)+\phi(x)) - CF(\phi(x-\Delta x) - 2\phi(x) + \phi(x+\Delta x)) & \text{if } u(x) > 0 \\ \frac{1}{2}(\phi(x+\Delta x)+\phi(x)) - CF(\phi(x) - 2\phi(x+\Delta x) + \phi(x+2\Delta x)) & \text{otherwise,} \end{cases} \tag{13}$$

where CF is a tuning coefficient.

```
float bho(Field b, int i, int j, int d1, int d2, float uf)
{
  // this function gets used in the advection() method in order to provide field values at
       the
  // faces of the staggered cells; the returned value is the value at the (i+d1,j+d2) face

  // b - the field to be interpolated
  // i,j - cell indices
  // d1,d2 - tell us on which face from i,j the values are being computed
  // uf - convection velocity on the face of interest

  // this part remains the same irrespectively of where the flow is coming from
  // note that it represents simple linear interpolation between cell i,j and i+d1,j+d2
  float bf =  0.5*(b.a[i+d1][j+d2]+b.a[i][j]);

  // check if the flow is in the negative direction - if yes then we need to
  // shift the stencil by one node UPWIND
  if (d1*uf<0)
  {
    i += d1;
    d1 = -d1;
  }

  if (d2*uf<0)
  {
    j += d2;
    d2 = -d2;
  }

  // if the stencil exceeds the grid dimensions then simply switch to a linear scheme
  if ( i>n-2 || i<2 || j>m-2 || j<2 ) return bf;

  // get the three values between which we want to fit a parabola
  float bc = b.a[i][j]; // value at the point of interest
  float bd = b.a[i+d1][j+d2]; // downwind value
  float bu = b.a[i-d1][j-d2]; // upwind value

  // bf is the typical QUICK implementation now, except the CF coefficient may be varied
  bf -= CF*(bd-2*bc+bu);

  // this is a test which approximates the face value by taking the upwind value,
  // and then going 10 d(phi)/dx from it
  float b1 = bu+S*(bc-bu);

  // this bounds the solution in some way between the cell value and a pre-set limit
  // with respect to the upwind cell
  return med(bf, bc,
    med(bc, bd, b1) // determine the upper bound for the limiter
    );
}
```

6.3.9 Advection

Compute the advection of a field b due to the velocity field described by this vector field object. The return value is the sum of fluxes of b into the ij cell. Need to distinguish whether the values of the b field are being stored on the negative x-faces or otherwise in order to construct appropriate staggered cells.

todo: would be good to put in figures showing the mesh and staggered cells, highlighting how the advection works; something like fig 6.3 on pp 126 from Murthy 2002 `https://engineering.purdue.edu/ME608/webpage/main.pdf`

```
float advection (Field b, int i, int j)
{
  // face velocity values - w,e,s,n
  float uo, ue, vs, vn;

  // get values on the faces
  if (b.btype == 1)
  {
    // if the values of the field to be convected are stored at the -ve x face we need
        convective
    // velocities (\delta x_i)/2 away in each direction from the vertical faces
    uo = 0.5*(x.a[i-1][j  ]+x.a[i  ][j  ]); // value at cell centre of (i-1,j)
    ue = 0.5*(x.a[i+1][j  ]+x.a[i  ][j  ]); // c.c. value at (i,j)
    vs = 0.5*(y.a[i  ][j  ]+y.a[i-1][j  ]); // value at SW corner of (i,j) cell
    vn = 0.5*(y.a[i  ][j+1]+y.a[i-1][j+1]); // value at NW corner of (i,j) cell
  }
  else
  {
    uo = 0.5*(x.a[i  ][j-1]+x.a[i  ][j  ]); // value at SW corner of (i,j) cell
    ue = 0.5*(x.a[i+1][j-1]+x.a[i+1][j  ]); // value at SE corner of (i,j) cell
    vs = 0.5*(y.a[i  ][j-1]+y.a[i  ][j  ]); // c.c. value in the S cell of (i,j)
    vn = 0.5*(y.a[i  ][j  ]+y.a[i  ][j+1]); // c.c. value in the (i,j) cell
  }
  // return the sum of fluxes of b INTO this cell - interpolate using QUICK scheme
  return ( (uo*bho(b, i, j, -1, 0, uo) - ue*bho(b, i, j, 1, 0, ue))
      + (vs*bho(b, i, j, 0, -1, vs) - vn*bho(b, i, j, 0, 1, vn)) );
}
```

6.3.10 Advection - diffusion

Combine the advection and diffusion terms and advance the flow field in time by a single time step. Takes the old velocity value, $u0$, computes the momentum flux into each cell, accounts for diffusion, and updates the velocity field using a first-order Euler time method. No external forces (like the pressure gradient) are accounted for.

```
void AdvDif(VectorField u0, float dt, float nu)
{
  VectorField v = new VectorField(this);
  for ( int j=1; j<m-1; j++)
  {
    for ( int i=1; i<n-1; i++)
    {
      v.x.a[i][j] = (advection(x, i, j) + nu*diffusion(x, i, j))*dt + u0.x.a[i][j];
      v.y.a[i][j] = (advection(y, i, j) + nu*diffusion(y, i, j))*dt + u0.y.a[i][j];
    }
  }
  this.eq(v);
}
```

6.3.11 Semi-Lagrangian advection

These two methods allow the components of the vector field to be advected with an external velocity field just as any scalar variable would.

```
// advect each of the components of this field as simple scalar variables given an
    external
// velocity field b; may use either 1st or 2nd order accuracy implemented in Field.pde
void advect( float dt, VectorField b )
{
  x.advect(dt,b);
  y.advect(dt,b);
}

void advect( float dt, VectorField b, VectorField b0 )
```

```
{
  x.advect(dt,b,b0);
  y.advect(dt,b,b0);
}
```

6.3.12 CFL

This computes a custom stability criterion, in a way similar to the Courant and Peclet numbers.

```
float CFL(float nu)
{
  // find maximum velocity
  float b = abs(x.a[0][0])+abs(y.a[0][0]);
  float c;
  for ( int i=1; i<n-1; i++)
  {
    for ( int j=1; j<m-1; j++)
    {
      c = abs(x.a[i][j])+abs(y.a[i][j]);
      if (c>b) b=c;
    }
  }
  return 1./(b+3.*nu);
}
```

6.3.13 Project

This accepts the coefficients on the left-hand side of the Poisson equation, computed in `BDIM::update()`, the pressure field from the previous time step, and the divergence of the recently computed velocity field. The latter may be viewed as the discrepancy between the desired divergence-free field and what was computed using the initial BDIM step. First, the Poisson equation is assembled into the matrix format. This is then passed to the multi-grid solver in order to find a pressure field whose gradient will correct the velocity in such a way as to enforce continuity, or

$$\Delta t \nabla \cdot \left(\frac{\mu_0^\epsilon}{\rho} \nabla p_0 \right) = \nabla \cdot \mathbf{U}' - (1 - \mu_0^\epsilon) \nabla \cdot \mathbf{U_B}. \tag{14}$$

the part associated with velocity divergence due the the body velocity gets omitted, any reason for this? maybe Lilypad only deals with solid bodies or there's a separate extension which incorporates this? Once this has been found, the step in equation 33 c in M&W 2015 may be undertaken, by first computing the pressure gradient and then evaluating the corrected, divergence-free velocity field

$$\mathbf{U_1} = \mathbf{U}' - \Delta t \frac{\mu_0^\epsilon}{\rho} \nabla p_0. \tag{15}$$

what's the p.sum() bit about?

```
// enforce continuity by solving the Poisson equation for this velocity field
Field project ( VectorField coeffs, Field p, Field s )
{
  /* projects u,v onto a divergence-free field using
  div{coeffs*grad{p}} = div{u}  (1)
  u -= coeffs*grad{p}           (2)
  and returns the field p. all FDs are on unit cells */

  // solves the equation of form Ax=b where A - Poisson matrix, b - s (div(U)), x - p
  p = MGsolver( 20, new PoissonMatrix(coeffs), p , s );

  // TODO what does this do? some sort of normalisation
  p.plusEq(-1*p.sum()/(float)((n-2)*(m-2)));

  // compute grad(p)
  VectorField dp = p.gradient();

  // velocity correction - 33c in M&W'15
  x.plusEq(coeffs.x.times(dp.x.times(-1)));
  y.plusEq(coeffs.y.times(dp.y.times(-1)));
  setBC();
  return p;
}
```

7 Poisson matrix

7.1 Description

This is used to assemble the Poisson equation of form

$$\nabla \cdot (C \nabla p) = b, \tag{16}$$

where b is a right-hand side that may incorporate various terms and C is a matrix of coefficients. This method is linked directly to the multigrid solver for all your computational needs.

Not entirely sure about some of the things, there should be more info here: `http://ocw.mit.edu/courses/mathematics/18-086-mathematical-methods-for-engineers-ii-spring-2006/readings/am35.pdf`

7.2 Internal fields

- `<int> n, m` - number of cells in the uniform grid
- `<VectorField> lower` - coefficients of the lower diagonal part of the equation
- `<Field> diagonal` - matrix with the diagonal terms
- `<Field> inv` - matrix filled with inverse values of the diagonal

7.3 Methods

7.3.1 Constructor

check what exactly sumd is

```
// construct given the coefficients matrix
PoissonMatrix( VectorField lower )
{
  this.n = lower.n;
  this.m = lower.m;
  this.lower = new VectorField(lower);
  diagonal = new Field(n,m);
  inv = new Field(n,m,0,1);

  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      // sum values at all faces of the cell and assign them to the diagonal
      float sumd = lower.x.a[i][j] + lower.x.a[i+1][j] + lower.y.a[i][j] + lower.y.a[i][j
          +1];
      diagonal.a[i][j] = -sumd;
      // compute inverse of the diagonal, be careful not to divide by zero
      if(sumd>1e-5) inv.a[i][j] = -1./sumd;
    }
  }
}
```

7.3.2 Multiplication

check what exactly this does

```
Field times( Field x )
{
  Field ab = new Field(n,m);
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      ab.a[i][j] = x.a[i  ][j  ]*diagonal.a[i][j] // multiply diagonal terms
            +x.a[i-1][j  ]*lower.x.a[i  ][j  ]
            +x.a[i+1][j  ]*lower.x.a[i+1][j  ]
            +x.a[i  ][j-1]*lower.y.a[i  ][j  ]
            +x.a[i  ][j+1]*lower.y.a[i  ][j+1];
    }
  }
  return ab;
}
```

7.3.3 Residual

Gets called from the MG solver only. check what exactly this does

```
Field residual( Field b, Field x ) {
  return b.plus(this.times(x).times(-1)); }
```

8 Boundary data immersion method - BDIM

8.1 Description

The core class in the code which implements the key functionality of the flow solver.

reference Maertens & Weymouth, "Accurate Cartesian-grid simulations of near-body flows at intermediate Reynolds numbers", 2015

The basic outline of using the class to solve a flow past an object is:

- Initialise a body and BDIM objects inside the `setup()` method in Processing
- Iterate in time using the `draw()` function, during which

  - Update position, velocity, and shape of the body (`Body::update()`)
  - Update coefficients due to the body, such as the wall-normal directions and distances, as well as BDIM kernels (`BDIM::update(body)`)
  - Still inside `BDIM::update()` advect the flow to obtain a new velocity field which does not necessarily satisfy continuity
  - Call `BDIM::updateUP()` in order to assemble the Poisson equation and solve it using `VectorField::project()`
  - Repeat the solution step again using `BDIM::update2()`, average the velocity fields obtained from the two updates in order to achieve second order accuracy in time (Euler integration with Heun's corrector)
  - Apply post-processing, move on to the next time step

8.2 Internal fields

- `<int> n, m` - number of cells in the uniform grid
- `<float> dt` - time step of the simulation
- `<float> nu` - dissipation constant, or viscosity
- `<float> eps` - support of the BDIM kernels, given a constant value of 2 by default
- `<PVector> g` - body force per unit mass (gravity), given a constant value of (0,0) by default
- `<VectorField>`
  - `u` - velocity field
  - `del` - zeroth moment of the smoothing kernel
  - `del1` - first moment of the smoothing kernel
  - `c` -
  - `u0` - velocity field at the previous time step
  - `ub` - body velocity field
  - `wnx, wny` - wall-normal direction to the nearest point on a body surface; these correspond to wall-normal x- and y-directions with their `VectorField::x` holding the actual values in x- and y-directions
  - `distance` - distance from the nearest body to the cell centres
  - `rhoi` - inverse of local fluid density (equal to 1 for single-phase flow)
- `<Field> p` - pressure field
- `<boolean> QUICK` - whether to solve the advection-diffusion problem (when set to true) or whether to use semi-Lagrangian convection
- `<boolean> mu1` - whether to use the first moment of the smoothing kernel, set as true by default
- `<boolean> adaptive` - whether to use adaptive time step

8.3 Methods

8.3.1 Constructor

The most important constructor. Several overloaded versions with simpler inputs also implemented.

check if x-velocity will ever have bval 0, otherwise what's the point of the gradientExit check?

```
BDIM( int n, int m, float dt, Body body, VectorField uinit, float nu, boolean QUICK )
{
  // set the constants
  this.n = n;
  this.m = m;
  this.dt = dt;
  this.nu=nu;
  this.QUICK=QUICK;

  u = uinit;

  if(u.x.bval!=0) u.x.gradientExit = true; // use gradient exit for the axial velocity
  u0 = new VectorField(n,m,0,0);
  p = new Field(n,m);
  if(dt==0) setDt(); // adaptive time stepping for O(2) QUICK

  ub  = new VectorField(n,m,0,0);
  distance =  new VectorField(n, m, 10, 10);
  del = new VectorField(n,m,1,1);
  del1 = new VectorField(n,m,0,0);
  rhoi = new VectorField(del); // initialise with ones
  c = new VectorField(del);
  wnx = new VectorField(n,m,0,0);
  wny = new VectorField(n,m,0,0);
  get_coeffs(body); // computes various coefficients for the flow
}
```

8.3.2 Get coefficients

Computes a range of coefficients needed to solve the flow.

```
void get_coeffs( Body body )
{
  get_dist(body); // distance from the body
  get_del(); // 0th BDIM kernel
  get_del1(); // 1st BDIM kernel
  get_ub(body); // body velocity
  get_wn(body); // wall-normal direction to the nearest body
}
```

8.3.3 Get distance

Calculates distance from the specified body to all of the cell centres.

```
void get_dist( Body body )
{
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      distance.x.a[i][j] = body.distance((float)(i-0.5), j);
      distance.y.a[i][j] = body.distance(i, (float)(j-0.5));
    }
  }
}
```

8.3.4 Get zeroth kernel moment

Compute the 0th order kernel moment for all cells given a body described using BDIM. This is called for each cell inside the `get_del()` method. As in ref, this is expressed as

$$\mu_0^\epsilon(d) = \begin{cases} \frac{1}{2}\left[1 + \frac{d}{\epsilon} + \frac{1}{\pi}\sin\left(\frac{d}{\epsilon}\pi\right)\right], & |d| < \epsilon, \\ 0, & d \le -\epsilon, \\ 1, & d \ge \epsilon, \end{cases} \tag{17}$$

where ϵ is a fixed kernel support width.

```
// calculate the 0th BDIM kernel value given a distance from the nearest body
float delta0( float d )
{
  if( d <= -eps ) // point inside the body
  {
    return 0;
  }
  else if( d >= eps ) // point well inside the flow
  {
    return 1;
  }
  else // point at the body-fluid interface, need to blend the solid and fluid equations
  {
    // eq. 16a in Maertens & Weymouth 2015
    return 0.5*(1.0+d/eps+sin(PI*d/eps)/PI);
  }
}
```

8.3.5 Get first kernel moment

Compute the 1st order kernel moment for all cells given a body described using BDIM. This is called for each cell inside the `get_del1()` method. As in ref, this is expressed as

$$\mu_1^\epsilon(d) = \begin{cases} \epsilon\left[\frac{1}{4} - \left(\frac{d}{2\epsilon}\right)^2 - \frac{1}{2\pi}\left(\frac{d}{\epsilon}\sin\left(\frac{d}{\epsilon}\pi\right) + \frac{1}{\pi}\left(1 + \cos\left(\frac{d}{\epsilon}\pi\right)\right)\right)\right], & |d| < \epsilon, \\ 0, & |d| \ge -\epsilon. \end{cases} \tag{18}$$

```
// compute the 1st BDIM kernel
float delta1( float d )
{
  if( abs(d) >= eps) // outside of the blending region
  {
    return 0;
  }
  else // inside the blending region use eq. 16b from Maertens & Weymouth 2015
  {
    return 0.25*(eps-sq(d)/eps)-1/TWO_PI*(d*sin(d*PI/eps)+eps/PI*(1+cos(d*PI/eps)));
  }
}
```

8.3.6 Body velocity

Computes velocity of the body, including translation and rotation, at each location within the numerical domain.

```
// calculates the velocity of the body at each location in the flow
void get_ub( Body body )
{
  /* Immersed Velocity Field
  ub(x) = U(x)*(1-del(x))
  where U is the velocity of the body */
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      ub.x.a[i][j] = body.velocity(1,dt,(float)(i-0.5),j);
      ub.y.a[i][j] = body.velocity(2,dt,i,(float)(j-0.5));
    }
  }
}
```

8.3.7 Wall-normal direction

Evaluate wall-normal direction to the nearest point on an immersed body.

check where this gets used and make sure the distances are from face centres

```
// computes wall normal direction of the closest body point
void get_wn(Body body)
{
  PVector wn;
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      wn = body.WallNormal((float)(i-0.5),j);
      wnx.x.a[i][j]=wn.x;
      wny.x.a[i][j]=wn.y;
      wn = body.WallNormal(i,(float)(j-0.5));
      wnx.y.a[i][j]=wn.x;
      wny.y.a[i][j]=wn.y;
    }
  }
}
```

8.3.8 Update

First, evaluates the constant coefficients that come into the Poisson equation,

$$c = \frac{\mu_0^\epsilon \Delta t}{\rho}, \tag{19}$$

thereby preparing for steps 33 b and 34 b in Maertens & Weymouth (2015). The current velocity field is then copied and stored in order to allow time interpolation later on. The flow equation of motion, rearranged for use in the BDIM method,

$$\mathbf{U}_0 + \mathbf{r}_{\Delta t}(\mathbf{U}) = \mathbf{U}_0 + \Delta t \left[-(\mathbf{U}\cdot\nabla)\mathbf{U} + \nu\nabla^2\mathbf{U} \right], \tag{20}$$

is then prepared using `VectorField::AdvDiff()`. This forms the first part of steps 33 a and 34 a in the M&W 2015. The `updateUP()` method is then called to solve the Poisson equation.

```
// advance the flow in time - 1st order time accuracy
void update()
{
  // O(dt,dx^2) BDIM projection step:
  c.eq(del.times(rhoi.times(dt))); // coefficient in front of the pressure gradient term
      in eq 33b and 34b in M&W'15

  u0.eq(u); // keep the current velocity for future use at thet next time step
  VectorField F = new VectorField(u); // this will hold the newly assembled flow equation
      of motion

  // either solve simple convection or full conv-diff problem
  if(QUICK)
    F.AdvDif( u0, dt, nu );
  else
    F.advect( dt, u0 );

  updateUP( F, c );
}
```

8.3.9 UpdateUP

First, the body force (not included in the original reference) is added to the right-hand side of equations 33a and 34b as

$$\mathbf{G} = \mathbf{g}\Delta t. \tag{21}$$

The result is then used to perform the first part of eq. 33 a,

$$\mathbf{U}' = \mu_0^\epsilon \left(\mathbf{U}_0 + \mathbf{r}_{\Delta t}(\mathbf{U})\right) - (1-\mu_0^\epsilon)\mathbf{U}_\mathbf{B}, \tag{22}$$

where $\mathbf{U}_\mathbf{B}$ is the body velocity. The first moment kernel contribution is then optionally added as

$$\mathbf{U}' + = \mu_1^\epsilon \frac{\partial}{\partial n} \left(\mathbf{U}_0 + \mathbf{r}_{\Delta t}(\mathbf{U}) - \mathbf{U}_\mathbf{B}\right). \tag{23}$$

This yields an updated velocity field which does not necessarily satisfy the continuity equation. This is then enforced by referring to `VectorField::project()` and solving the Poisson equation. Note that until now the effect of pressure gradient was ignored, analogously to how M&W manipulate the governing flow equations between eq. 26 and 33. This is done in a similar fashion to the typical predictor-corrector approach for finite volume CFD.

```
void updateUP( VectorField R, VectorField coeff )
{
  /*  Seperate out the pressure from the forcing
    del*F = del*R+coeff*gradient(p)
  Approximate update (dropping ddn(grad(p))) which doesn't affect the accuracy of the
      velocity
    u = del*R+coeff*gradient(p)+[1-del]*u_b+del_1*ddn(R-u_b)
  Take the divergence
    div(u) = div(coeff*gradient(p)+stuff) = 0
  u.project solves this equation for p and then projects onto u
  */
  R.plusEq(PVector.mult(g,dt)); // assemble the mu_0 term with u^0 in eq. 33a in M&W 2015;
       add body force
  u.eq(del.times(R).minus(ub.times(del.plus(-1)))); // 1st order BDIM using only zeroth
      moment - first part of eq 33a
  if(mu1) u.plusEq(del1.times((R.minus(ub)).normalGrad(wnx,wny))); // add the 2nd order
      terms - full eq. 33a
  u.setBC(); // set inflow/outflow and other BCs
  // make sure the velocity field is divergence free by adjusting the pressure correcition
       - eqns. 33b and 33c
  // the overloaded implementation of project is called which neglects the divergence
      which could
  // be created by changes to the geometry of the immersed body
  // TODO why?
  p = u.project(coeff,p);
}
```

8.3.10 Update2

This method gets called after `update()` and is used in order to ensure second-order accuracy in time. In general, it involves identical steps to the first-order update step, except the initial condition is not the velocity from the old time step but the one obtained during the initial predictor-corrector solution. Once the new velocity field satisfying continuity has been estimated, two solutions are effectively available:

$$\begin{aligned} \mathbf{U}_1(t_0+\Delta t) &= f(\mathbf{U}_0(t), t+\Delta t), \\ \mathbf{U}_2(t_0+\Delta t) &= f(\mathbf{U}_1(t+\Delta t), t+\Delta t). \end{aligned} \tag{24}$$

Heun's approach involves using the average of the two estimates to be the value at the new time step.

```
// advance the flow in time - 2nd order time accuracy
void update2()
{
  // O(dt^2,dt^2) BDIM correction step:
  // us holds the divergence-free velocity field obtained during the first Euler step,
      therefore 1st order
  // accurate in time
  VectorField us = new VectorField(u);
  VectorField F = new VectorField(u);

  // this uses the QUICK interpolation on a segregated mesh to solve the advection-
      diffusion problem
  if(QUICK)
  {
    // evaluate the flow using BDIM starting from the velocity computed during the first
        step
    F.AdvDif( u0, dt, nu );
    updateUP( F, c );

    // Heun's corrector - take average of U(U_0) and U(U_1)
    u.plusEq(us);
    u.timesEq(0.5);

    // adjust time step to maintain stability
    if(adaptive) dt = checkCFL();
  }
  // this advects each component of this field as a simple scalar variable
  else
  {
    F.eq(u0.minus(p.gradient().times(rhoi.times(0.5*dt))));
    F.advect(dt,us,u0);
    updateUP( F, c.times(0.5) );
  }
}
```