LilyPad base class reference

November 1, 2015

Abstract

Somebody elses code is always hard, let's break it down!

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` - type of boundary condition; possible values
  - 0 - zero gradient (Dirichlet) BC in all directions

  - 1 - fixed value (Neumann) BC in the x-direction
  - 2 - Neumann BC in the y-direction

- `<float> bval` - value of the boundary condition (only used in certain cases of `btype` values)
- `<bool> gradientExit` - flag specifying whether gradient outlet BC is used; important only if `btype` is 1 or 2

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 1st order backward 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
      // 1st order backward scheme
      // 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
  // 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 makes the lhs and bottom
  // inlets for u and v, and rhs and top outlets for u and v, respectively
  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 we are using Neumann BC (1 or 2) - we want to keep interpolation
      // points to be cell centres and so we need to fool the interpolation method
      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), interpoalte the field values in space to get a value at that location. Why do the quadratic and linear interpolation methods increment x and y by a half if the BC is specified to Neumann in the respective direction? Wherever they're actually used this seems to be compensated for by calling them with x-0.5 in the first place

```
float quadratic( float x0, float y0)
{
  // correct (x0,y0) to allow for BCs
  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.

Boundary types:

- 0 - zero gradient (Dirichlet) applied everywhere (take 0th and N-1st cell in x and y and give them a value equal to that one cell into the domain (1st and N-2nd)
- 1 - fixed value (Neumann) type BC in the x-direction
  - If `gradientExit` is specified true then apply the `bvalue` to the 1st (not 0th!) cell in the x-direction across all y-values; then, a mean value at the outflow is computed using N-2 index (note how actually index N-1 is used but that value has just been assigned N-2 value a few lines above). Now we let all N-1 cells have the value of `bval` less the mean. why subtract the mean?
  - Otherwise, let all cells at 1st and N-1st index in the x-direction equal to `bval`. The value is assigned at N-1st index not to interfere with zero gradient condition later correct?. why assign to N-1st and not N-2nd then?
- 2 - this is analogous to `btype` 1 but in the y-direction; in this case there is no `gradientExit` option and so `bval` is assigned to cells both at the top and at the bottom of the domain (again, use 1st and N-1st)

```
// correct the boundary conditions
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 value zeros
    // 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(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 1-into the domain (not 0 and n-1)
      // TODO why?
      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 x-direction is fixed-value then assign the 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 Divergence

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

$$\nabla \cdot \mathbf{U}. \tag{5}$$

The derivative operator is approximated using first order forward 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++ )
    {
      d.a[i][j] = x.a[i+1][j  ]-x.a[i][j]+
            y.a[i  ][j+1]-y.a[i][j];
    }
  }
  return d;
}
```

6.3.2 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

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

where the overline denotes an ensemble average. why add phi_i and phi_(i+1)? and what about the -2?

```
Field ke ()
{
  Field d = new Field( n, m );
  for ( int i=1 ; i<n-1 ; i++ )
  {
    for ( int j=1 ; j<m-1 ; j++ )
    {
      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;
}
```

6.3.3 Vorticity

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

$$\nabla \times \mathbf{U}. \tag{7}$$

Gradients are computed using central scheme. both gradients are computed at (i,j) and (i+1,j+1) and then averaged, why?

```
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 magnitude
      // gradients approximated using central difference
      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 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 j+1
    }
  }
  return d;
}
```

6.3.4 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? 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++ )
    {
      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;
}
```

6.3.5 Diffusion

Use second order linear scheme to compute the 2nd spatial derivative around cell. In one dimension this could be written as

$$\frac{\partial^2\phi}{\partial x^2} \approx \phi(x-\Delta x) - 2\phi(x) + \phi(x+\Delta x) + O[\Delta x^2]. \tag{11}$$

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

$$\nu\nabla^2\mathbf{U} = \frac{\partial^2 u_k}{\partial x_i x_j}\hat{\mathbf{e}}_k\delta_{ij} = \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)\mathbf{i} + \left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)\mathbf{j}. \tag{12}$$

Note that this function computes the Laplacian for a single component of a vector field only.

```
float diffusion (Field b, int i, int j)
{
  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];
}
```

6.3.6 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)));
}
```

6.3.7 Face value interpolation

why use CF=1/6 in QUICK if most people quote it as 1/8?

why the name "bho"?

Why the med limiter is called with boundaries which seem to be the cell value bc as long as bc is between the downwind and "test" upwind value?

In order to compute sum of the fluxes on each cell face, the cell-centre values need to be interpolated. This function makes use of the quadratic upwind scheme. For example, for the east face of a cell P located at x this would be

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

QUICK, in order to accomplish this.

```
float bho(Field b, int i, int j, int d1, int d2, float uf)
{
  // 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]; // cell value
  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
    );
}
```

6.3.8 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.

why distinguish based on the inlet which values to interpolate?

```
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 inlet is in the x-direction
  if (b.btype == 1)
  {
    uo = 0.5*(x.a[i-1][j  ]+x.a[i  ][j  ]);
    ue = 0.5*(x.a[i+1][j  ]+x.a[i  ][j  ]);
    vs = 0.5*(y.a[i  ][j  ]+y.a[i-1][j  ]);
    vn = 0.5*(y.a[i  ][j+1]+y.a[i-1][j+1]);
  }
  else
  {
    uo = 0.5*(x.a[i  ][j-1]+x.a[i  ][j  ]);
    ue = 0.5*(x.a[i+1][j-1]+x.a[i+1][j  ]);
    vs = 0.5*(y.a[i  ][j-1]+y.a[i  ][j  ]);
    vn = 0.5*(y.a[i  ][j  ]+y.a[i  ][j+1]);
  }
  // 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)) );
}
```

6.3.9 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. why use 2nd order RK in scalars but 1st order here?

```
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);
}
```

6.3.10 CFL

is this really a CFL number if it doesn't involve the time step and we take the inverse of maximum velocity? why include viscosity? why compute maximum velocity as u+v and not $sqrt(u^2 + v^2)$?

```
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);
}
```