Velocity Reconstruction, Interpolation and Computing Back-Trajectories: Requirements and Design

MPAS Development Team

November 5, 2012

Contents

Chapter 1

Summary

The use of space-time transport algorithms, such Characteristic Discontinuous Galerkin (CDG) or Incremental Remapping (IR), requires the computation of Lagrangian pre-images. These pre-images are formed by tracing velocities defined at the vertices of the finite volume cells backward in time. In addition, some space-time algorithms such as CDG need to compute forward trajectories from arbitrary points within finite-volume cells.

The use of space-time transport algorithms requires the ability to 1) reconstruct full velocity vectors at vertices of finite-volume cells, 2) compute back-trajectories of these full velocity vectors and 3) the ability to determine the velocity vector at any point within a finite volume cell. MPAS-Ocean currently lacks the ability to do 1, 2 or 3. The code currently uses radial-basis functions to reconstruct velocity vectors at cell centers and this could readily be generalized to cell vertices.

This development contains three main challenges. First, we must construct full velocity vectors using nearby velocity component data. Second, we need to insure that back trajectories land in the ocean and not within land. And third, we need a sensible way to interpolate velocity vectors to any point within an ocean cell.

Ultimately we want this to work on the sphere, but this document is restricted to planar meshes as a starting point.

Success is as follows: Given the normal components of a linearly-varying velocity field at C-grid velocity points, the method exactly reconstructs this vector field at vertices and computes exact back trajectories. In addition, the vector velocity interpolation method at any point within the cell is exact for linearly-varying velocity vectors. This all points to a second-order accurate scheme.

Chapter 2

Requirements

2.1 Requirement: Vector Reconstruction at Vertices

Date last modified: 2012/11/01
Contributors: (Todd Ringler)

The reconstruction of vector at vertices of finite volume cells (aka Voronoi cells) should be exact for linearly varying velocity fields. If we are given a velocity field

$$\mathbf{P}(\mathbf{x}) = (u(\mathbf{x}), v(\mathbf{x})) = (c_0 + c_1 x + c_2 y, d_0 + d_1 x + d_2 y) \tag{2.1}$$

we compute the normal component of $\mathbf{P}$ at C-grid velocity points as

$$u_n(\mathbf{x}_e) = \mathbf{P}(\mathbf{x}_e) \cdot \mathbf{n}(\mathbf{x}_e) \tag{2.2}$$

where $\mathbf{x_e}$ are the location of C-grid velocity points and $\mathbf{n}_e$ are unit vectors normal to cell edges. Now using as much or as little $u_n(\mathbf{x}_e)$ data as we choose, we compute

$$\mathbf{R}(\mathbf{x}_v) = f(u_n(\mathbf{x}_e)) \tag{2.3}$$

where $\mathbf{R}(\mathbf{x}_v)$ the reconstruct vector velocity field at vertices based on f, the yet-to-be-determined reconstruction function. The requirement is that

$$\mathbf{R}(\mathbf{x}_v) = \mathbf{P}(\mathbf{x}_v) \tag{2.4}$$

given (2.1). The above is just a fancy way of saying that we can exactly reconstruct linearly-varying velocity fields.

2.2 Requirement: Vector Interpolation

Date last modified: 2012/11/01
Contributors: (Todd Ringler)

The vector interpolation of vector velocity at an arbitrary point within a Voronoi cells should be exact for a linearly-varying velocity field. Assuming that Requirement 2.1 is met, then we know the reconstructed velocity at cell vertices, $\mathbf{R}(\mathbf{x}_v)$, is exact for linearly-varying functions. For an interpolated velocity, $\mathbf{I}$, evaluated at an arbitrary point, $\mathbf{x}_i$ within a cell, the requirement for interpolation is

$$\mathbf{I}(\mathbf{x}_i) = \mathbf{P}(\mathbf{x}_i) \tag{2.5}$$

when the velocity has the form given in (2.1). The interpolated velocity can be constructed based on $\mathbf{R}(\mathbf{x}_e)$ or $u_n(\mathbf{x}_e)$ or both.

2.3 Requirement: Back Trajectory Calculation

Date last modified: 2012/11/01
Contributors: (Todd Ringler)

Given $\mathbf{R}(\mathbf{x}_v, t^n, t^{n+1})$ and a time step, $dt = t^{n+1} - t^n$, the method is required to find the point $\mathbf{x}_D(t^n)$

$$\mathbf{x}_D(t^n) = \mathbf{x}_v - \int_{t^{t+1}}^{t^n} \mathbf{I}(\mathbf{x}_i, t^n, t^{n+1}) dt \tag{2.6}$$

where $\mathbf{x}_i$ is along the back-trajectory characteristic and $\mathbf{x}_v$ is the location of a Voronoi vertex.

2.3.1 Accuracy

The accuracy requirement is that $\mathbf{x}_D(t^n)$ is exact for linearly-varying vector velocity fields. The notion of linearly-varying is extended here to include the time dimension.

2.3.2 Bounds

The bounds requirement is that $\mathbf{x}_D(t^n)$ is guaranteed to reside within the ocean domain.

Chapter 3

Algorithmic Formulations

3.1 Design Solution: Vector Reconstruction at Vertices

Date last modified: 2011/01/05
Contributors: Todd Ringler and Pedro de Silva Peixoto

The vector reconstruction at cell vertices will be conducted using a least-squares solution using nearby normal-component velocity data points (reference Vidovic). Figure 3.1 shows a portion of a Voronoi mesh. The solid circle at the vertex (call it V) is the location at which we wish to do a reconstruction. The algorithm will use the velocity component data along the edges connected to V and the data along the edges connected to those edges. This results in 9 data values (nData=9) to be used in the reconstruction. Near boundaries, the recovery of 9 data values in a symmetric fashion is not possible. In those cases, we simply use the 9 closest velocity points. Each of these velocity data points are associated with a normal vector (shown in red in Figure 3.1 to be referred to as $\mathbf{n}_k(x_k, y_k)$, where k ranges from 1 to nData.

$$\begin{pmatrix} n_{x,1} & n_{x,1}x_1 & n_{x,1}y_1 & n_{y,1} & n_{y,1}x_1 & n_{y,1}y_1 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ n_{x,k} & n_{x,k}x_k & n_{x,k}y_k & n_{y,k} & n_{y,k}x_k & n_{y,k}y_k \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ d_0 \\ d_1 \\ d_2 \end{pmatrix} = \begin{pmatrix} u_{n,1} \\ \vdots \\ u_{n,k} \end{pmatrix} \quad (3.1)$$

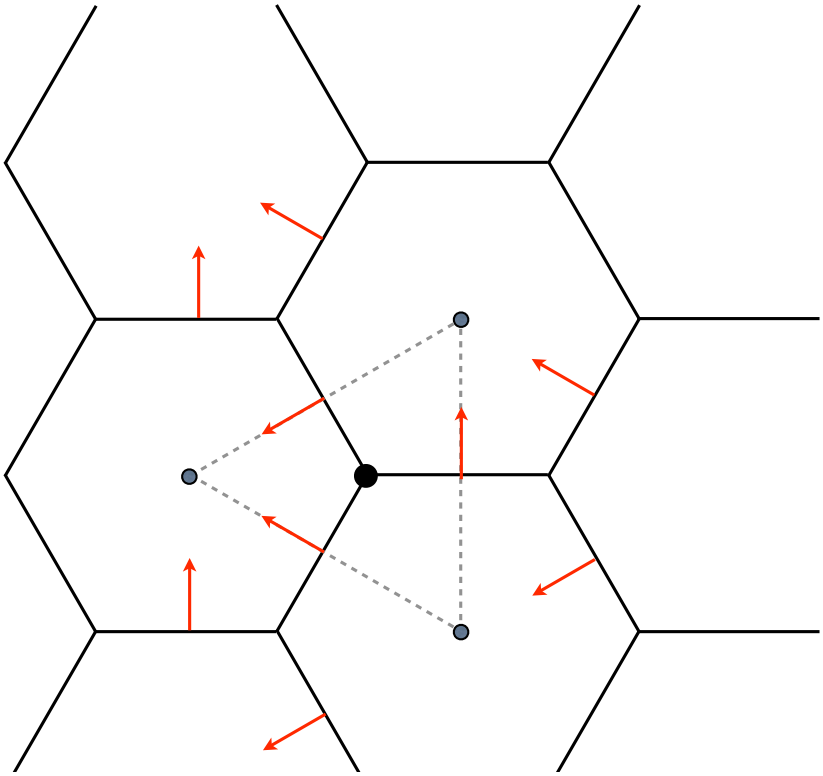

Figure 3.1: Data used in least-squares reconstruction.

where the RHS is the normal-component velocity data. If we call the LHS matrix to be $\mathbf{M}$, tnhe least-squares solution is of the form

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ d_0 \\ d_1 \\ d_2 \end{pmatrix} = \left[\left(\mathbf{M}^T \mathbf{M} \right)^{-1} \mathbf{M}^T \right] \begin{pmatrix} u_{n,1} \\ \vdots \\ u_{n,k} \end{pmatrix}. \tag{3.2}$$

The matrix $\left[\left(\mathbf{M}^T \mathbf{M} \right)^{-1} \mathbf{M}^T \right]$ will be computed at start up and stored at each vertex. If we set up a local coordinate system at each vertex with $(0, 0)$ residing at the $\mathbf{x}_v$, then the reconstructed velocity is simply

$$\mathbf{R}(\mathbf{x}_v) = (c_0, d_0) \tag{3.3}$$

MATLAB code to test this least-squares reconstruction is here:
https://www.dropbox.com/s/69ql731ke1x8mwy/leastSquaresReconstruction.zip

3.2 Design Solution: Vector Interpolation

Date last modified: 2011/01/05
Contributors: Todd Ringler and Pedro de Silva Peixoto

The velocity vector within a cell will be expressed as a function of the vector velocities at the cell vertices, i.e.

$$\mathbf{I}(\mathbf{x}_i) = \sum_{k=1}^{nVert} \lambda_k \mathbf{R}_k \tag{3.4}$$

where $\mathbf{x}_i$ is a point within the Voronoi cell, $\mathbf{R}_k$ is the set of vector velocities residing at vertices and λ_j is the interpolation weight.

The interpolation weights are derived using Wachspress coordinates (reference Wachspress 1975 and Gillette paper), also known as generalized barycentric coordinates. As shown in Figure 3.2, each vertex will be associated with an area $B_k = T(\mathbf{x}_{k-1}, \mathbf{x}_k, \mathbf{x}_{k+1})$ where T is the area of a triangle based on the location of its vertices. Note that B_k is not a function of the interpolation point $\mathbf{x}_i$. Given an interpolation point $\mathbf{x}_i$, we now define $nVert$ triangles as $A_k = T(\mathbf{x}_k, \mathbf{x}_{k+1}, \mathbf{x}_i)$. The Wachspress coordinate is then

$$w_k(\mathbf{x}_i) = B_k \prod_{l \neq k, l \neq k-1} A_l(\mathbf{x}_i) \tag{3.5}$$

and the interpolation weight for each vertex is then

$$\lambda_k(\mathbf{x}_i) = \frac{w_k(\mathbf{x}_i)}{\sum_{l=1}^{nVert} w_l(\mathbf{x}_i)} \tag{3.6}$$

The λ functions can be thought of as non-negative, basis functions for each vertex. An example of one of these basis function is shown in Figure 3.3. These basis functions have some very desirable properties for our application. First, $\lambda_k(\mathbf{x}_{v,j}) = 1$ for $k = j$ and zero otherwise. Along an edge connecting two vertices, say k and $k+1$, the interpolation is a linear function of λ_k and λ_{k+1} only. This results in $\lambda_k = 0$ along all edges not connected to $\mathbf{x}_{v,k}$. The interpolation is exact for linear functions (reference). And finally, if we impose no-slip boundary conditions as $\mathbf{R}_k = 0$ at boundary vertices, this zero-velocity condition holds exactly along all boundary edges. (Note: A special fix will be needed for those parts of the ocean domain that are only one-edge).

MATLAB code to test Wachspress coordinates is here:
https://www.dropbox.com/s/oggd4iyekeqg3ng/WachspressCoordinates.zip

3.3 Design Solution: Backward Trajectories

Date last modified: 2011/01/05
Contributors: Todd Ringler and Rob Lowrie

The proposed algorithm is a self-correcting, mid-point rule for the integration along particle trajectories. The "self-correcting" nature of the algorithm is that if any points along the particle trajectory resides outside the ocean domain, the algorithm "reboots" automatically in order to do a more accurate trajectory calculation. By virtue of the mid-point (trapezoidal) integration, the scheme will be exact for linearly-varying flows in space and time. The algorithm outlined in Algorithm 1 depends on the vector interpolation scheme described above in order to compute velocities at arbitrary locations within a Voronoi cell. Algorithm 1 meets the accuracy and bounds requirements listed above.

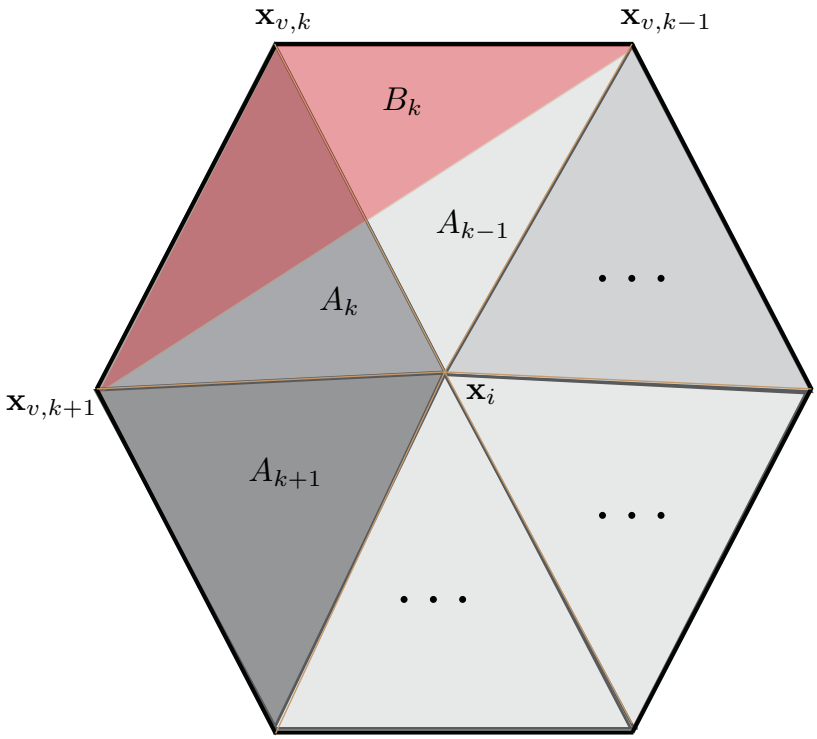

Figure 3.2: Interpolation of data to a point $\mathbf{x}_i$ based on vertex data defined at $\mathbf{x}_v$.

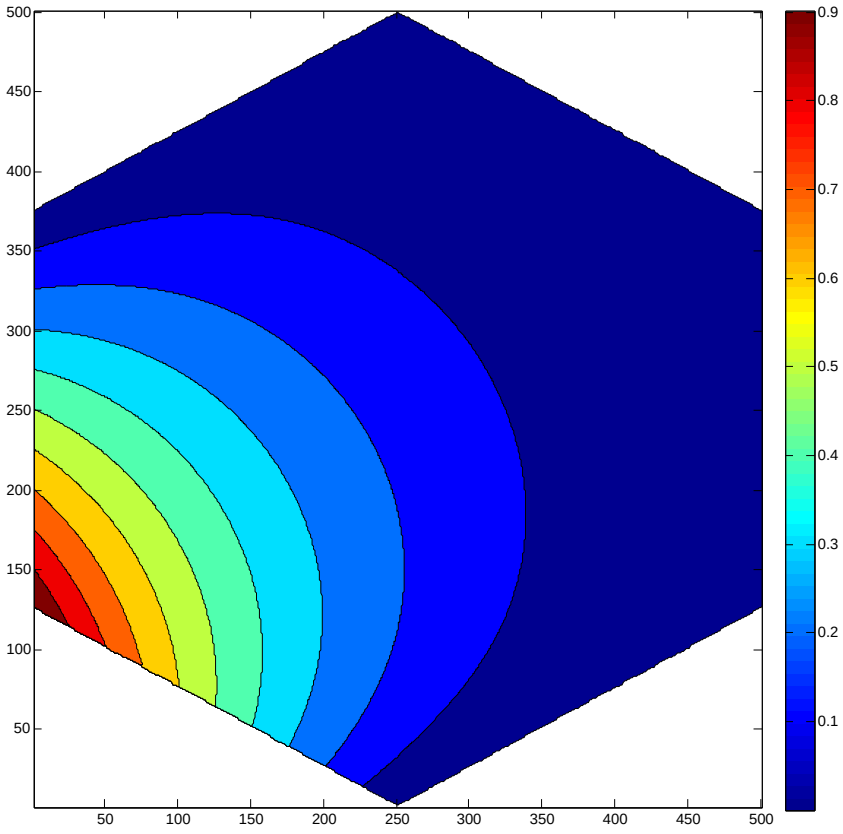

Figure 3.3: Structure of a Wachspress basis function.

3.4 Design Solution: Forward Trajectories

Date last modified: 2011/01/05
Contributors: Todd Ringler and Rob Lowrie
The proposed algorithm is identical to the backward trajectory Algorithm 1 with two exceptions. First, the negative sign used in the backward trajectory to indicate backward-in-time is replace with a positive sign. This occurs on line 22 and 32. Second, the starting location and velocity is the set to the quadrature point and velocity at quadrature point instead of the vertex location and vertex velocity. Given the similarity between the forward and backward trajectory calculation, it is anticipated that both will be done with exactly the same source code.

Algorithm 1 Calculation of backward particle trajectories

This algorithm is executed at every Voronoi vertex
The algorithm should be valid for arbitrarily large CFL
kLevel is the vertical layer in which to compute back trajectories
$?_A$, $?_D$ and $?_{mid}$ are Arrival, Departure and mid-point *on each segment*
nSegments is the number of line segments to be used in back trajectory
nSegmentsMin will typically be 1
nSegments=max(nSegmentsLastTime-1,nSegmentsMin)-1
kTest = -1 ▷ Set to false so that do While loop is entered
$\mathbf{u}_A = \mathbf{u}_D = \mathbf{u}_v^{n+1}$ ▷ Initialize A and D velocities to be vertex velocity
$\mathbf{x}_D = \mathbf{x}_v$ ▷ Initialize departure point to be vertex location
for While kTest < 0 **do**
    kTest=1 ▷ Assume it all works out
    nSegments=nSegments+1
    dtSegment = dt / nSegments
    for iSegment=1,nSegments **do** ▷ Integrate over segments
        $t_D = t^{n+1}$-iSegment*dtSegment
        $t_A = t_D$+dtSegment
        $\mathbf{u}_A = \mathbf{u}_D$ ▷ The arrival velocity is previous departure velocity
        $\mathbf{x}_A = \mathbf{x}_D$ ▷ The arrival point is previous departure point
        $\mathbf{u}_{mid} = \mathbf{u}_D$ ▷ Initialize midpoint velocity
        for iter=1,nIterMidPoint **do** ▷ Iterate to find midpoint values
            $\mathbf{x}_D = \mathbf{x}_A$-dtSegment*$\mathbf{u}_{mid}$
            find iCellBase; $\mathbf{x}_D$ resides in iCellBase
            kTest = min(maxLevelCell(iCellBase)-kLevel,kTest)
            Linearly interpolate $\mathbf{u}_v^n$ and $\mathbf{u}_v^{n+1}$ to $\mathbf{u}_v^n(t_D)$
            $\mathbf{u}_D$=vectorInterpolate($\mathbf{x}_D, \mathbf{u}_v$(iCellBase,$t_D$))
            $\mathbf{x}_{mid} = 1/2(\mathbf{x}_D + \mathbf{x}_A)$
            find iCellBase; $\mathbf{x}_{mid}$ resides in iCellBase
            kTest = min(maxLevelCell(iCellBase)-kLevel,kTest)
            $\mathbf{u}_{mid}$=vectorInterpolate($\mathbf{x}_{mid}, \mathbf{u}_v$(iCellBase,$t_{mid}$))
        end for ▷ iter
        $\mathbf{x}_D = \mathbf{x}_A$-dtSegment*$\mathbf{u}_{mid}$
    end for ▷ iSegment
    if nSegments>nSegmentsMax then STOP
end for ▷ kTest
nSegmentsLastTime=nSegments

Chapter 4

Design and Implementation

4.1 Implementation: XXX

Date last modified: 2011/01/05
Contributors: (add your name to this list if it does not appear)

This section should detail the plan for implementing the design solution for requirement XXX. In general, this section is software-centric with a focus on software implementation. Pseudo code is appropriate in this section. Links to actual source code are appropriate. Project management items, such as svn branches, timelines and staffing are also appropriate.

How do we typeset pseudo code?

```
verbatim?
```

Chapter 5

Testing

5.1 Testing and Validation: XXX

Date last modified: 2011/01/05
Contributors: (add your name to this list if it does not appear)

How will XXX be tested? i.e. how will be we know when we have met requirement XXX. Will these unit tests be included in the ongoing going forward?