1 Capabilities

This version of flowVC is capable of reading in 2D or 3D velocity data and

1. Computing FTLE fields,
2. Computing tracer/particle trajectories,
3. Interpolating the velocity data onto another mesh.

You cannot compute FTLE and particle trajectories at the same time. The feature to interpolate velocity data onto another mesh is mainly useful to check if the program is reading in and interpolating your velocity data correctly. The velocity data must be defined on a Cartesian OR tetrahedral (3D) / triangular (2D) mesh.

2 Running flowVC

flowVC is a command line program. All parameters are set in an input file. The file name is a command line argument, and the file must be located in the directory you are calling the program from. Like any executable, flowVC should exist in the calling directory, or in any executable path (e.g., /usr/local/bin/).

```
$ flowVC FTLE-example.in
```

3 Format of input files

At a minimum, you will need to provide flowVC with velocity data. To keep the code (and data files) "light-weight," flowVC only inputs (and outputs) binary data files. The standard practice is to write a small program that converts your data into the right format for flowVC. Here we describe that format. Generally, you will need separate files defining the velocity field at each time point, and a file (Cartesian data) or files (unstructured data) defining the mesh. All files should have a common prefix of your choosing, listed here generically as `InFilePrefix`.

3.1 Files defining velocity data

If the velocity is defined discretely in time, there should be a separate file for each time instant. Also, the velocity data must be equally spaced in time. The files should be named `InFilePrefix_vel.#.bin`, where # is a unique index for each time instant. This index does not need to increment by 1, but the increment between successive files must remain fixed. The content of the file is a time stamp (`double`) followed by the velocity vectors at each node (3 `doubles`). You can envision the file looking like what is shown below. For Cartesian data, the ordering should be such that the x-direction is looped over in the *inner-most* loop, then y, then z in the outer loop. For unstructured data, the ordering should be consistent with the ordering of the nodes in the `InFilePrefix_coordinates.#.bin` file (see below).

Note: For 2D data, you still need to specify a third component of velocity, w_i, which can be set to 0.0.

`InFilePrefix_vel.#.bin`:
$ts \quad u_0 \quad v_0 \quad w_0 \quad u_1 \quad v_1 \quad w_1 \quad \dots \quad u_{(n-1)} \quad v_{(n-1)} \quad w_{(n-1)}$

3.2 File(s) defining mesh data

3.2.1 Cartesian

For velocity data defined on a Cartesian mesh, only the mesh bounds and resolution need to be defined in a file `InFilePrefix_Cartesian.bin`. The resolutions are the number of nodes in each direction, not intervals, and are type `int`. The bounds are type `double`. Note: For 2D data, set $zmin = zmax = 0.0$ and $zres = 1$.

`InFilePrefix_Cartesian.bin`:
$xmin \ \ xmax \ \ xres \ \ ymin \ \ ymax \ \ yres \ \ zmin \ \ zmax \ \ zres$

3.2.2 Unstructured

For velocity data defined on a unstructured mesh, three separate files are needed:
`InFilePrefix_coordinates.bin`, `InFilePrefix_connectivity.bin`,
`InFilePrefix_adjacency.bin`.

The file `InFilePrefix_coordinates.bin` list the number of nodes n (`int`) and the coordinates of each node (3 `doubles`). Note: For 2D data, you still need to specify a third component of the position, z_i, which can be set to 0.0.

`InFilePrefix_coordinates.bin`:
$n \quad x_0 \quad y_0 \quad z_0 \quad x_1 \quad y_1 \quad z_1 \quad \dots \quad x_{(n-1)} \quad y_{(n-1)} \quad z_{(n-1)}$

The file `InFilePrefix_connectivity.bin` list the number of elements e (`int`) and the connectivity for each element (4 `ints`). The connectivity specifies the indices of the 4 nodes that make up the element. Numbering of the nodes is implied by the ordering in `InFilePrefix_coordinates.bin`, counting from zero. Note: For 2D data, you still need to specify a fourth node index, n_i^3, which can be set to -1.

`InFilePrefix_connectivity.bin`:
$e \quad i_0^0 \quad i_0^1 \quad i_0^2 \quad i_0^3 \quad i_1^0 \quad i_1^1 \quad i_1^2 \quad i_1^3 \quad \dots \quad i_{(e-1)}^0 \quad i_{(e-1)}^1 \quad i_{(e-1)}^2 \quad i_{(e-1)}^3$

The file `InFilePrefix_adjacency.bin` list the number of elements e (`int`) and the adjacency for each element (4 `ints`). The adjacency specifies the indices of each of the 4 elements that share a common face with the element. Numbering of the elements is implied by the ordering in `InFilePrefix_connectivity.bin`, counting from zero. For boundary elements,

which have faces not shared with any other element, the index should be set to -1. Note: For 2D data, you still need to specify a fourth element index, e_i^3, which can be set to -1. The ordering of the adjacency is important; see Appendix below.

`InFilePrefix_adjacency.bin`:
$e \quad e_0^0 \quad e_0^1 \quad e_0^2 \quad e_0^3 \quad e_1^0 \quad e_1^1 \quad e_1^2 \quad e_1^3 \quad \ldots \quad e_{(e-1)}^0 \quad e_{(e-1)}^1 \quad e_{(e-1)}^2 \quad e_{(e-1)}^3$

4 Format of output files

The output files have a similar binary format to the input files described above. Nominally, field information is listed in separate files for each time instant, and the content of each file is a time stamp followed by the field values. Corresponding mesh files are generated as well, which depend on if the data is Cartesian or unstructured. For tracer/particle position output, there are no associated mesh files; each file contains a time stamp followed by the list of coordinates for the tracer positions at the time stamp. Some examples are below.

`FTLE.#.bin`:
$ts \quad \lambda_1 \quad \lambda_2 \quad \ldots \quad \lambda_{(n-1)}$

`FTLE_Cartesian.bin`:
$xmin \;\; xmax \;\; xres \;\; ymin \;\; ymax \;\; yres \;\; zmin \;\; zmax \;\; zres$

—
`tracers.#.bin`:
$ts \quad x_0 \quad y_0 \quad z_0 \quad x_1 \quad y_1 \quad z_1 \quad \ldots \quad x_{(n-1)} \quad y_{(n-1)} \quad z_{(n-1)}$

Provided with flowVC is a program, bin2vtk, that converts various types of data output from flowVC into (legacy) vtk ASCII format, which can be read into ParaView. Paraview is a free and open source visualization program `http://www.paraview.org`. To see the usage of bin2vtk just call the program with no command line arguments, as shown below:

```
$ bin2vtk

Usage:
  bin2vtk DataType ND FilePrefix Start End Delta (MeshFilePrefix) (-flags)

Description:
  Converts FilePrefix.#.bin to FilePrefix.#.vtk,
  where # varies from Start to End in increments of Delta.
  ND should 2 or 3 depending if data is 2D or 3D
  flags:
  -i: used for integer valued data
  -s: used if tracer data has scalar values assigned to each tracer

  Supported values for DataType:
  0: Tracer position data
```

```
  1: Scalar unstructured node data*
  2: Vector unstructured node data*
  3: Scalar unstructured element data*
  4: Vector unstructured element data*
  5: Scalar Cartesian node data**
  6: Vector Cartesian node data**
  7: Scalar Cartesian element data**
  8: Vector Cartesian element data**
  *Requires files MeshFilePrefix_coordinates.bin and MeshFilePrefix_connectivity.bin
  **Requires file MeshFilePrefix_Cartesian.bin
```

E.g., to convert FTLE results for 2D flow (e.g., the double gyre example) you would enter something like:

```
$ bin2vtk 5 2 dg_forwardFTLE 0 10 1 dg_forwardFTLE
```

and the output should look like

```
Converting dg_forwardFTLE.0.bin to dg_forwardFTLE.0.vtk...OK!
Converting dg_forwardFTLE.1.bin to dg_forwardFTLE.1.vtk...OK!
Converting dg_forwardFTLE.2.bin to dg_forwardFTLE.2.vtk...OK!
Converting dg_forwardFTLE.3.bin to dg_forwardFTLE.3.vtk...OK!
Converting dg_forwardFTLE.4.bin to dg_forwardFTLE.4.vtk...OK!
Converting dg_forwardFTLE.5.bin to dg_forwardFTLE.5.vtk...OK!
Converting dg_forwardFTLE.6.bin to dg_forwardFTLE.6.vtk...OK!
Converting dg_forwardFTLE.7.bin to dg_forwardFTLE.7.vtk...OK!
Converting dg_forwardFTLE.8.bin to dg_forwardFTLE.8.vtk...OK!
Converting dg_forwardFTLE.9.bin to dg_forwardFTLE.9.vtk...OK!
Converting dg_forwardFTLE.10.bin to dg_forwardFTLE.10.vtk...OK!
```

After this you should be able to load the data into Paraview and visualize the results.

Appendix

The adjacency should be sorted to make local search faster, see Shadden, Astorino & Gerbeau (2010) `http://mmae.iit.edu/shadden/_media/shaddenastorinogerbeau10.pdf`.

Here is section of code that maps element to natural coordinates and updates test element to neighbor if point is not in the original test element.

```
/* Physical coordinates of nodes of the test element */
x0 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[0]][0];
y0 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[0]][1];
z0 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[0]][2];
x1 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[1]][0];
y1 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[1]][1];
z1 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[1]][2];
x2 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[2]][0];
y2 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[2]][1];
z2 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[2]][2];
x3 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[3]][0];
y3 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[3]][1];
z3 = Vel_MeshNodeArray[Vel_MeshElementArray[pt->ElementIndex].Nodes[3]][2];

/* Entries for mapping of physical to natural coordinates of test element */
a11 = (z3 - z0) * (y2 - y3) - (z2 - z3) * (y3 - y0);
a21 = (z3 - z0) * (y0 - y1) - (z0 - z1) * (y3 - y0);
a31 = (z1 - z2) * (y0 - y1) - (z0 - z1) * (y1 - y2);
a12 = (x3 - x0) * (z2 - z3) - (x2 - x3) * (z3 - z0);
a22 = (x3 - x0) * (z0 - z1) - (x0 - x1) * (z3 - z0);
a32 = (x1 - x2) * (z0 - z1) - (x0 - x1) * (z1 - z2);
a13 = (y3 - y0) * (x2 - x3) - (y2 - y3) * (x3 - x0);
a23 = (y3 - y0) * (x0 - x1) - (y0 - y1) * (x3 - x0);
a33 = (y1 - y2) * (x0 - x1) - (y0 - y1) * (x1 - x2);

/* Determinant of mapping */
V = (x1 - x0) * ((y2 - y0) * (z3 - z0) - (z2 - z0) * (y3 - y0)) +
  (x2 - x0) * ((y0 - y1) * (z3 - z0) - (z0 - z1) * (y3 - y0)) +
  (x3 - x0) * ((y1 - y0) * (z2 - z0) - (z1 - z0) * (y2 - y0));

/* Natural coordinates of point to be interpolated */
r = (a11 * (x - x0) + a12 * (y - y0) + a13 * (z - z0)) / V;
s = (a21 * (x - x0) + a22 * (y - y0) + a23 * (z - z0)) / V;
t = (a31 * (x - x0) + a32 * (y - y0) + a33 * (z - z0)) / V;

/* Check if point inside element */
d = fmin(r, fmin(s, fmin(t, 1 - r - s - t)));
if(d < 0) {
  /* Not inside, reset test element to appropriate neighbor */
  if(fabs(r - d) < TINY)
    pt->ElementIndex = Vel_MeshElementArray[pt->ElementIndex].NeighborIndex[0];
  else if(fabs(s - d)  < TINY)
    pt->ElementIndex = Vel_MeshElementArray[pt->ElementIndex].NeighborIndex[1];
  else if(fabs(t - d) < TINY)
    pt->ElementIndex = Vel_MeshElementArray[pt->ElementIndex].NeighborIndex[2];
  else if(fabs((1 - r - s - t) - d) < TINY)
    pt->ElementIndex = Vel_MeshElementArray[pt->ElementIndex].NeighborIndex[3];
  else
    FatalError("Indeterminate neighbor in function GetVelocity_Unstructured()");
 }
```

You can see the last part resets the guess element to one of the neighbors, depending on which condition was most violated. Therefore, the neighbors need to be sorted and the sorting depends on the mapping. Since the mapping is hard coded, the neighbors are always sorted the same way and listed in `InFilePrefix_adjacency.bin`. For example, here's the snippet of code that sorts the neighbors properly when generating the `_adjacency.bin` file from a program we use called vis2bin, which as you may guess converts from vis file format (output from CFD solver) to bin file format needed for flowVC.

```
for(i = 0; i < Mesh_NumElements; i++) {
  face1[0] = Element_Array[i].Nodes[0];
  face1[1] = Element_Array[i].Nodes[2];
  face1[2] = Element_Array[i].Nodes[3];
  face2[0] = Element_Array[i].Nodes[0];
  face2[1] = Element_Array[i].Nodes[1];
```

```
  face2[2] = Element_Array[i].Nodes[3];
  face3[0] = Element_Array[i].Nodes[0];
  face3[1] = Element_Array[i].Nodes[1];
  face3[2] = Element_Array[i].Nodes[2];
  face4[0] = Element_Array[i].Nodes[1];
  face4[1] = Element_Array[i].Nodes[2];
  face4[2] = Element_Array[i].Nodes[3];
  indexf1 = -1;
  indexf2 = -1;
  indexf3 = -1;
  indexf4 = -1;
  /* Loop over neighbors */
  for(j = 0; j < NumNeighbors[i]; j++) {
    /* Check for shared face */
    if(listcmp(Element_Array[Element_Array[i].NeighborIndex[j]].Nodes, face1, 4, 3) == 3)
      indexf1 = Element_Array[i].NeighborIndex[j];
    else if(listcmp(Element_Array[Element_Array[i].NeighborIndex[j]].Nodes, face2, 4, 3) == 3)
      indexf2 = Element_Array[i].NeighborIndex[j];
    else if(listcmp(Element_Array[Element_Array[i].NeighborIndex[j]].Nodes, face3, 4, 3) == 3)
      indexf3 = Element_Array[i].NeighborIndex[j];
    else if(listcmp(Element_Array[Element_Array[i].NeighborIndex[j]].Nodes, face4, 4, 3) == 3)
      indexf4 = Element_Array[i].NeighborIndex[j];
    else
      FatalError("face not matched");
  }
  /* Sort */
  Element_Array[i].NeighborIndex[0] = indexf1;
  Element_Array[i].NeighborIndex[1] = indexf2;
  Element_Array[i].NeighborIndex[2] = indexf3;
  Element_Array[i].NeighborIndex[3] = indexf4;
}

---

int listcmp(int *list1, int *list2, int list1_length, int list2_length) {
  /*** Returns number of matching members in the two lists ***/
  int i, j, match;
  match = 0;
  for(i = 0; i < list1_length; i++)
    for(j = 0; j < list2_length; j++)
      if(list1[i] == list2[j])
        match++;
  return(match);
}
```