Assignment #7
3D Vector Field Visualization—Streamlines and Stream Ribbons
Due Oct 30th before midnight

Goal:
In this assignment, you are required to implement a number of visualization techniques to help explore a number of 3D steady vector fields provided below, including arrow plots, streamlines, stream ribbons, and an animated 3D probe.

Tasks:
Similar to Assignment #4, you need to create a regular grid with the range. You can reuse the same data structure but include additional member variables to store the three components of the vector value.

-1 \leq x, y, z \leq 1

Compute a vector value at each node point according to the following formulas for the X, Y, and Z components of the vector based on the following formula

\begin{align*}
\text{field 1:} & \\
v_x(x, y, z) &= -3 + 6x - 4x(y + 1) - 4z \\
v_y(x, y, z) &= 12x - 4x^2 - 12z + 4z^2 \\
v_z(x, y, z) &= 3 + 4x - 4x(y + 1) - 6z + 4(y + 1)z
\end{align*}

\begin{align*}
\text{field 2:} & \\
v_x(x, y, z) &= A \sin z + C \cos y \\
v_y(x, y, z) &= B \sin x + A \cos z \\
v_z(x, y, z) &= C \sin y + B \cos x
\end{align*}

where \( A = \sqrt{3}, B = \sqrt{2}, C = 1 \)

\begin{align*}
\text{field 3:} & \\
v_x(x, y, z) &= -y \\
v_y(x, y, z) &= -z \\
v_z(x, y, z) &= x
\end{align*}

The following routine can be used to compute the vector value at any given 3D position for field 1. Note that you will be using this function to get the vector value at any spatial location for your streamline computation as well.

void get_vector_field1( float x, float y, float z, float &vxp, float &vyp, float &vzp )
{
    vxp = -3 + 6.*x - 4.*x*(y+1.) - 4.*z;
    vyp = 12.*x - 4.*x*x - 12.*z + 4.*z*z;
    vzp = 3. + 4.*x - 4.*x*(y+1.) - 6.*z + 4.*(y+1.)*z;
}

JS Version
function get_vector_field1(x, y, z)
{
    var vxp = -3 + 6.*x - 4.*x*(y+1.) - 4.*z;
}
Var vyp = 12.*x - 4.*x*x - 12.*z + 4.*z*z;
Var vzp = 3. + 4.*x - 4.*x*(y+1.) - 6.*z + 4.*(y+1.)*z;
Return [vxp, vyp, vzp]

Please define two additional functions similar to the above one for the other two fields.

1. Direct visualization (20 points)

Assign a 3D arrow for each grid point. The shape of the arrow is up to you. But you can always use the one provided in the skeleton code. Note that you need to scale the arrows uniformly in order to get reasonable visualization as you did in Assignment #6. The color of each arrow is determined by the magnitude of the corresponding vector value. You can use any color scales that you have implemented in Assignment #2.

2. Streamlines (50 points)

Draw streamlines within the field starting with at least 10 distinct points (i.e., at least 10 streamlines should be computed). My suggestion is to try to start with a streamline for each grid point and see whether you can get a reasonable visualization. Then, try to place the seeds for those streamlines smartly. How to determine the places of seeding is up to you.

Please use at least two different numerical integration techniques to compute streamlines. One of them has to be second-order RK integrator. Please include an interface for the user to choose one of the integration schemes.

Allow the user to move a 3D streamline probe through the volume. Every time the probe moves, its streamline needs to be re-computed. How you position the starting position of the probe in X-Y-Z is up to you. I will suggest starting from the origin (0, 0, 0). To move the starting position, modify the translation widget in the interface. (If you use GLUI interface, you can use three Translate widgets to allow translation in XY, XZ, and YZ; If you are using AntTweakBar, include three writable variables that enable users to input three values to move the starting position, and a button to update the new streamline(s)).

3. Stream ribbon (30 points)

Compute and render a stream ribbon starting at any given position. Allow the user to move the ribbon probe through the volume, similar to what you do for the streamline probe. Instead of having a single streamline coming from the probe, attach a small horizontal line to the probe and have some number of stream lines coming from different places on the line. Connect them up
with GL_QUADS. It is not required to do OpenGL lighting (but it is a nice touch). Please see slide 20 in Lecture 13 for the detailed algorithm of computing a streamline ribbon.

**Grades:**

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Total points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
</tr>
</tbody>
</table>

**Suggestions:**

Draw each streamline until: 1) It leaves the bounding cube given above, or, 2) The change in $(x,y,z)$ becomes nearly zero (i.e., vector magnitude is close to zero), or, 3) You have drawn for a certain number of steps (try a couple hundred)

If you do not have a routine for 3D arrow drawing, you can use the following piece of code to generate 3D arrows by Prof. Mike Bailey.

```c
/**********************************************************************
/* Global variables for axes, by Mike Bailey */
 /**********************************************************************
/* size of wings as fraction of length: */
#define WINGS 0.10

/* axes: */
#define X 1
#define Y 2
#define Z 3

/* x, y, z, axes: */
static float axx[3] = { 1., 0., 0. };  
static float ayy[3] = { 0., 1., 0. };  
static float azz[3] = { 0., 0., 1. };  

/////Function for axes drawing, borrow from arrow.c
void Arrow( float tail[3], float head[3] )
{
    float u[3], v[3], w[3];   /* arrow coordinate system */
    float d;   /* wing distance */
    float x, y, z;   /* point to plot */
    float mag;   /* magnitude of major direction */
    float f;   /* fabs of magnitude */
```
int axis;          /* which axis is the major */

/* set w direction in u-v-w coordinate system: */
w[0] = head[0] - tail[0];

/* determine major direction: */
axis = X;
mag = fabs( w[0] );
if( (f=fabs(w[1])) > mag )
{
    axis = Y;
mag = f;
}
if( (f=fabs(w[2])) > mag )
{
    axis = Z;
mag = f;
}

/* set size of wings and turn w into a unit vector: */
d = WINGS * unit( w, w );

/* draw the shaft of the arrow: */
glBegin( GL_LINE_STRIP );
glVertex3fv( tail );
glVertex3fv( head );
glEnd();

/* draw two sets of wings in the non-major directions: */
if( axis != X )
{
    cross( w, axx, v );
    (void) unit( v, v );
cross( v, w, u );
x = head[0] + d * ( u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
    glVertex3fv( head );
    glVertex3f( x, y, z );
    glEnd();
    x = head[0] + d * ( -u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
    glVertex3fv( head );
    glVertex3f( x, y, z );
    glEnd();
}
if( axis != Y )
{
    cross( w, ayy, v );
    (void) unit( v, v );
    cross( v, w, u );
    x = head[0] + d * ( u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
        glVertex3fv( head );
        glVertex3f( x, y, z );
    glEnd();
    x = head[0] + d * ( -u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
        glVertex3f( x, y, z );
    glEnd();
}

if( axis != Z )
{
    cross( w, azz, v );
    (void) unit( v, v );
    cross( v, w, u );
    x = head[0] + d * ( u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
        glVertex3f( head );
        glVertex3f( x, y, z );
    glEnd();
    x = head[0] + d * ( -u[0] - w[0] );
    glBegin( GL_LINE_STRIP );
        glVertex3f( x, y, z );
    glEnd();
}
/* done: */

//calculate the dot production of two vectors
float dot( float v1[3], float v2[3] )
{
    return( v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2] );
}
///calculate the cross production of two vectors
void cross( float v1[3], float v2[3], float vout[3] )
{
    float tmp[3];
    tmp[1] = v2[0]*v1[2] - v1[0]*v2[2];
    tmp[2] = v1[0]*v2[1] - v2[0]*v1[1];
    vout[0] = tmp[0];
    vout[1] = tmp[1];
}

//Normalize vector
float unit( float vin[3], float vout[3] )
{
    float dist, f ;
    dist = vin[0]*vin[0] + vin[1]*vin[1] + vin[2]*vin[2];
    if( dist > 0.0 )
    {
        dist = sqrt( dist );
        f = 1. / dist;
        vout[0] = f * vin[0];
        vout[1] = f * vin[1];
        vout[2] = f * vin[2];
    }
    else
    {
        vout[0] = vin[0];
        vout[1] = vin[1];
        vout[2] = vin[2];
    }
    return( dist );
}

Javascript version

By using Javascript, you can implement a version similar C++. Here is an example:

/* size of wings as fraction of length: */
const WINGS = 0.10;

/* axes: */
const X = 1;
const Y = 2;
const Z = 3;
/* x, y, z, axes: */
var axx = [ 1., 0., 0. ];
var ayy = [ 0., 1., 0. ];
var azz = [ 0., 0., 1. ];

/**
 * Draw a 3D arrow given the 3D positions of the tail and head
 * @param {*} tail an array with 3 values
 * @param {*} head an array with 3 values
 * @param {*} color an array with 4 values (RGBA)
 * @param {*} modelViewMatrix
 * @param {*} projectionMatrix
 */

function Arrow( tail, head, color, modelViewMatrix, projectionMatrix)
{
    var u = new Array();
    var v = new Array();
    var w = new Array(); /* arrow coordinate system */
    var d; /* wing distance */
    var x, y, z; /* point to plot */
    var mag; /* magnitude of major direction */
    var f; /* fabs of magnitude */
    var axis; /* which axis is the major */

    /* set w direction in u-v-w coordinate system: */
    w[0] = head[0] - tail[0];

    /* determine major direction: */
    axis = X;
    mag = Math.abs( w[0] );
    if( (f=Math.abs(w[1])) > mag )
    {
        axis = Y;
        mag = f;
    }
    if( (f=Math.abs(w[2])) > mag )
    {
        axis = Z;
        mag = f;
    }
}
} /* set size of wings and turn w into a unit vector: */
d = WINGS * unit( w, w )[1];

// Create a buffer for the vertex positions.
const vertexBuffer = gl.createBuffer();

// Select the positionBuffer as the one to apply buffer
// operations to from here out.

gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
var vertices = [];

/* draw the shaft of the arrow: */
/* draw two sets of wings in the non-major directions: */
if( axis != X )
{
  var dist;
  v = cross( w, axx);
  v, dist = unit( v, v );
  u = cross( v, w);
  x = head[0] + d * ( u[0] - w[0] );
  // Put coordinates of head and x,y,z computed above in your vertex buffer

  x = head[0] + d * ( -u[0] - w[0] );
  // Put coordinates of head and x,y,z computed above in your vertex buffer

}

if( axis != Y )
{
  var dist;
  v = cross( w, ayy);
  v, dist = unit( v, v );
  u = cross( v, w);
  x = head[0] + d * ( u[0] - w[0] );
  // Put coordinates of head and x,y,z computed above in your vertex buffer

}
// Put coordinates of head and x,y,z computed above in your vertex buffer

    x = head[0] + d * ( -u[0] - w[0] );

    if( axis != Z )
    {
        var dist;
        v = cross( w, azz, v );
        v, dist = unit( v, v );
        u = cross( v, w);
        x = head[0] + d * ( u[0] - w[0] );
        var colors = [
            [color[0], color[1], color[2],color[3]],
            [color[0], color[1], color[2],color[3]],
            [color[0], color[1], color[2],color[3]],
            [color[0], color[1], color[2],color[3]],
            [color[0], color[1], color[2],color[3]],
        ];
    }

    /* done: */

     gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(vertices), gl.STATIC_DRAW);

    const lineColors = [
        [color[0], color[1], color[2],color[3]],
        [color[0], color[1], color[2],color[3]],
        [color[0], color[1], color[2],color[3]],
        [color[0], color[1], color[2],color[3]],
        [color[0], color[1], color[2],color[3]],
        ];
for (var j = 0; j < lineColors.length; ++j) {
    const c = lineColors[j];

    colors = colors.concat(lineColors[j]);
}

const colorBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, colorBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(colors), gl.STATIC_DRAW);

// Build the element array buffer; this specifies the indices
// into the vertex arrays for each line's vertices.
const indexBuffer = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, indexBuffer);
const indices = [
    0, 1,
    2, 3,
    4, 5,
    6, 7,
    8, 9
];

// Now send the element array to GL

gl.bufferData(gl.ELEMENT_ARRAY_BUFFER,
    new Uint16Array(indices), gl.STATIC_DRAW);

const buffers = {
    position: vertexBuffer,
    color: colorBuffer,
    indices: indexBuffer,
};

// Draw buffers to the screen by using modelMatrix and projectionMatrix
}

//calculate the dot production of two vectors
function dot(v1, v2) {
    return ( v1[0]*v2[0] + v1[1]*v2[1] + v1[2]*v2[2] );
}
calculate the cross production of two vectors

```javascript
function cross( v1, v2 )
{
    var tmp = new Array();
    var vout = new Array();
    tmp[1] = v2[0]*v1[2] - v1[0]*v2[2];
    tmp[2] = v1[0]*v2[1] - v2[0]*v1[1];

    vout[0] = tmp[0];
    vout[1] = tmp[1];

    return vout;
}
```

Normalize vector

```javascript
function unit( vin )
{
    var dist, f;
    var vout = new Array();

    dist = vin[0]*vin[0] + vin[1]*vin[1] + vin[2]*vin[2];

    if( dist > 0.0 )
    {
        dist = Math.sqrt( dist );
        f = 1.0 / dist;
        vout[0] = f * vin[0];
        vout[1] = f * vin[1];
        vout[2] = f * vin[2];
    }
    else
    {
        vout[0] = vin[0];
        vout[1] = vin[1];
        vout[2] = vin[2];
    }

    return [vout, dist];
}
```
Example outputs for field 1 (the other two fields should have similar visualizations but with different flow patterns)

Arrow plot

Streamlines

Stream ribbon

Multiple streamlines

In both cases, the seeding position is highlighted as a small block. But you can use other shape to indicate the location of the seeding position.
The following is a reference interface design. Feel free to create your own layout that you prefer.