A Visualization of Vector Fields

A.1 Notation

Vectors are a set of objects that exist in a *vector space* V. On the vector space, there are defined two operations; addition and multiplication[1]. The entire vector space is spanned by the orthogonal basis vectors.

Definition. *Let the orthoganal basis* $\{\hat{e}_1, \hat{e}_2, \hat{e}_3\}$ *span vector space V, then a linear combination of the bases represent a vector* $\mathbf{v}$ *in* $\mathbf{R}^3$*. In component form, the vector is given as*

$$\mathbf{v} = v_1\hat{e}_1 + v_2\hat{e}_2 + v_3\hat{e}_3. \tag{1}$$

The components of the vector $\mathbf{v}$ can be, for the sake of brevity, written as a tuple (v_1, v_2, v_3) with the understanding[**?**] that we are refering to the component form (1). Likewise, we can define a vector in $\mathbf{R}^N$, where the N components in component form, are given as $(v_1, \ldots, v_N)$. Dots imply the remaining components of the vector, instead of listing them all up. The vector space is spanned by the basis $\{\hat{e}_1, \ldots, \hat{e}_N\}$.

The length of an N-dimensional vector is given by the Euclidean norm $\|\mathbf{v}\|$

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \ldots + v_N^2}\,. \tag{2}$$

Given a vector $\mathbf{v}$, we can normalize it by dividing it by it's own length $\mathbf{v}/\|\mathbf{v}\|$. Such a normalized vector is called a *unit vector.* If a set of unit vectors are orthogonal to each other, then they are called *orthonormal.* If a set of orthonormal vectors span the entire vector space V, then such a set is called *orthonormal basis.* Unless we specify otherwise, all bases herein will be assumed to be orthonormal.

Assigning a vector to each point in a subset of space generates a *vector field.* This requires a slightly different notation than Equation(1). In order to assign a location for a vector (v_1, v_2, v_3), we consider the components of the vector as a function of $\mathbf{x}$, where $\mathbf{x} = (x_1, x_2, x_3)$.

$$\mathbf{v} = v_1(\mathbf{x})\hat{e}_1 + v_2(\mathbf{x})\hat{e}_2 + v_3(\mathbf{x})\hat{e}_3.$$

As with Equation(1), we can extend this to N-dimensional vector spaces. The vector components would then be functions of $\mathbf{x} = (x_1, \ldots, x_N)$. A simple vector field (x, y, z) is displayed in Figure 1. Here, the magnitude of the vectors is displayed by color. As we would expect, for a cartesian coordinate system, the "intensity" of the field increases the further we get away from the origo.

A.2 Streamlines

Besides using vectors to display vector fields, another useful method is the displaying of streamlines.

[1] Along with the axioms that must hold for all vectors in V.

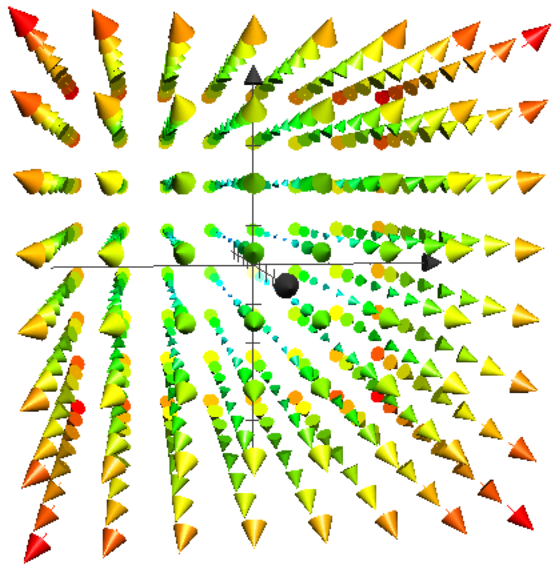

Figure 1: Vector field (x,y,z) displayed using a simple graphical tool in OS X.

Definition. *A Streamline is a curve where the vectors in a vector field are always tangent to the curve.*

For a vector field we can display the movement of a given particle at any location in the vector field. If the vector field changes with time, i.e vector field is unsteady, then we freeze the time at an instaneous moment, and draw the streamlines. The streamline is the path which the particle takes, or rather the path it must take as determined by the vector field at an instantaneous moment[1]. Mathematically, we can determine the path by solving the differential equations

$$\left.\frac{dx_1}{v_1(\mathbf{x})}\right|_{t=t_0} = \left.\frac{dx_2}{v_2(\mathbf{x})}\right|_{t=t_0} = \left.\frac{dx_3}{v_3(\mathbf{x})}\right|_{t=t_0} \tag{3}$$

By integrating separably, these equations can be solved at any position $\mathbf{x}$ and a given time $t = t_0$.

Numerically, this is accomplished by creating a grid for the vector field. Thereupon, numerical integration is performed on the computational grid. Using for example the forward Euler scheme, we can advance the path which the particle takes by a user-defined step size. The process of solving the system of ODE (ordinary differential equations) by a finite difference method is as following.

We start at a given point in the domain and allow the vector field to dictate in which direction and to where we are to progress along. Hence, the path becomes the unknown quantity which we must determine. For $\mathbf{v} \in \mathbf{R}^2$, we can list the whole process :

[1]For a steady field, the vector field never changes. Hence the streamlines will thus remain constant.

1. Discretize the domain $\mathbf{x}\in[0,\mathrm{x}_{\mathrm{max}}]\times[0,\ \mathrm{y}_{\mathrm{max}}]$ where the vector field exists.
2. Use a start point $(\mathrm{x}_0,\mathrm{y}_0)$ to initiate the integration.
3. Perform forward and backward integration (depending on where you start in the domain).

To perform the numerical integration, there are several numerical methods that can be employed.

A.3 Numerical Integration

The following algorithm lists all the necessary steps involved when performing numerical integration

---

Algorithm 1 Integration by forward Euler scheme

Load vector field data into arrays
Create a grid over a domain for which the vector field exists upon
Create arrays for storage of the streamline data
Perform numerical integration by the forward Euler method
Repeat process for backwards integration
Stitch together all the entries for forward and backwards integration

---

The computational grid, defined at postion (x_i, y_j), can be drawn as following

$$\begin{array}{ccc} \overset{\bullet}{i-1, j+1} & \overset{\bullet}{i, j+1} & \overset{\bullet}{i+1, j+1} \\ \overset{\bullet}{i-1, j} & \overset{\bullet}{i, j} & \overset{\bullet}{i+1, j} \\ \overset{\bullet}{i-1, j-1} & \overset{\bullet}{i, j-1} & \overset{\bullet}{i+1, j-1} \end{array}$$

where $i, j \in \mathbf{N}$. As a streamline is integrated in between grid points, we have to use interpolation of the surrounding gridpoints, where the values that are interpolated are the vectors. We can accomplish this by performing a bilinear interpolation, or an even higher order interpolation.

B Eigendecomposition

A vector $\mathbf{v}$ in $\mathbf{R}^N$ (where $N \geq 2$) is an *eigenvector* of a square $(NtimesN)$ matrix T, if

$$T\mathbf{v} = \lambda\mathbf{v},$$

where λ is called the *eigenvalue*. In order to determine the eigenvalue, we can solve the eigenvalue equation

$$det(T - \lambda I) = 0,$$

where I is the identity matrix, with the same dimension as the matrix T. For each unique eigenvalue λ_i, we can solve the eigenvalue equation

$$(T - \lambda_i I)\mathbf{v_i} = \vec{0}.$$

The eigenvectors $\mathbf{v}_i$ are mutually orthogonal. For the stress tensor τ_{ij}, the eigenvector of τ are called *pricipal axes.*

We can always factorize a diagonalizable[1] square $(N \times N)$ matrix with N eigenvectors

$$T = P^{-}1LU^{-}1,$$

where the column i of U is the eigenvector $\mathbf{v_i}$, and L is a diagonal matrix, with the eigenvalues $L_{ii} = \lambda_i$.

C Code Listings

C.1 Locate Degenerate Points

degeneracies.py

```python
import numpy as np

def T1(x_):
    x = x_[0]
    y = x_[1]
    return np.array([[.5*x**2 + 2*x*y + .5*y**2, -x**2 + y**2],
                     [ -x**2 + y**2, -.5*x**2 - 2*x*y - 5*y**2]])
def T2(x_):
    x = x_[0]
    y = x_[1]
    return np.array([[x**2 - y**2, 2*x*y],
                     [2*x*y, x**2 - y**2]])
def T3(x_):
    x = x_[0]
    y = x_[1]
    return np.array([[x**2 - 3*y**2, -5*x*y + 4*y**2],
                     [-5*x*y + 4*y**2, x**2 - 3*y**2]])
def T4(x_):
    x = x_[0]
    y = x_[1]
    return np.array([[x**4 - .5*x**2*y**2, 2*x**4 - 5*x**3*y - 9*x*y**3],
                     [2*x**4 - 5*x**3*y - 9*x*y**3, x**4 - .5*x**2*y**2]])

def T5(x_):
    x = x_[0]
    y = x_[1]
    return np.array([[-x**2 + y**2, -x**2 - 2*x*y + y**2],
                     [-x**2 - 2*x*y + y**2, -x**2 + y**2]])

def degenerate(T,tol=1e-12):
    """
    Function assumes that the tensor is given with the following form :

    If the tensor values are distributed over a 100x100 grid, then for 2D,
    (100,100,2,2) is the shape of the tensor T. Meaning, each grid point
    contains a 2 dimensional second rank tensor. We assume that all such
    tensors are symmetric. For a 3D tensor defined over a 16x50x16 grid,
```

[1]A non diagonalizable matrix, is a defective matrix that does not have a complete basis of eigenvectors.

```
    T is of the shape (16,50,16,3,3). Here too we assume that all grid
    tensors are symmetric.
    """
    if T.shape[-2:] == (2,2):
        if len(T.shape[:-2]) == 2:
            deg_points = _find_all_degen_points(T,dim=2,tol=tol)
            return deg_points
    elif T.shape[-2:] == (3,3):
        if len(T.shape[:-2]) == 3:
            deg_points = _find_all_degen_points(T,dim=3,tol=tol)
            return deg_points
    msg = "Tensor has wrong shape.\n"
    msg += "Tensor shape must be either in 3D\n"
    msg += "(data x, data y, data z, gridx, gridy, gridz)\n"
    msg += "or in 2D\n(data x, data y, gridx, gridy)"
    raise IndexError(msg)

def _find_all_degen_points(T,dim,tol):
    if tol>1e-6:
        print "Warning : Tolerance value provided is too large."
    xdata = T.shape[0]
    ydata = T.shape[1]
    if dim == 3:
        zdata = T.shape[2]
    fdp = 0 # counter for the amount of times we find a degenerate point

    class SparseMatrix:
        def __init__(self):
            self.entries = {}

        def __call__(self, tuple, value=0):
            self.entries[tuple] = value

        def value(self, tuple):
            try:
                value = self.entries[tuple]
            except KeyError:
                value = 0
            return value

    deg_points = SparseMatrix()
    if dim == 2:
        """
        Solve following system of equations for a 2D tensor :
            T[i,j,0,0] - T[i,j,1,1] == 0
            T[i,j,0,1] == 0
        """
        found = np.zeros(2)
        for i in range(xdata):
            for j in range(ydata):
                found[0] = np.fabs(T[i,j,0,0] - T[i,j,1,1]) <= tol
                found[1] = np.fabs(T[i,j,0,1]) <= tol

                if np.all(found):
                    deg_points((i,j,k),1)
                    fdp = fdp + 1
    elif dim == 3:
        """
        Solve following system of equations for a 3D tensor :
            T[i,j,k,0,0] - T[i,j,1,1] == 0
            T[i,j,k,1,1] - T[i,j,2,2] == 0
            T[i,j,k,0,1] == 0
            T[i,j,k,0,2] == 0
            T[i,j,k,1,2] == 0
        """
        found = np.zeros(5)
        for i in range(xdata):
            for j in range(ydata):
```

```python
            for k in range(zdata):
                found[0] = np.fabs(T[i,j,k,0,0] - T[i,j,k,1,1]) <= tol
                found[1] = np.fabs(T[i,j,k,1,1] - T[i,j,k,2,2]) <= tol
                found[2] = np.fabs(T[i,j,k,0,1]) <= tol
                found[3] = np.fabs(T[i,j,k,0,2]) <= tol
                found[4] = np.fabs(T[i,j,k,1,2]) <= tol

                if np.all(found):
                    deg_points((i,j,k),1)
                    fdp = fdp + 1

    print "Found %d degenerate points in the tensor data." %(fdp)
    return deg_points, fdp
```

C.2 Invariance

invariance.py

```python
import sympy
from sympy import diff
import numpy

def msg(case,T,x):
    dim = len(x)
    T = numpy.array(T)
    if case:
        print "\nThe tensor"
        print T
        if dim==2:
            print "has a degenerate point at (%d,%d)" %(x[0],x[1])
        if dim==3:
            print "has a degenerate point at (%d,%d,%d)" %(x[0],x[1],x[2])
    else:
        print "\nThe Tensor"
        print T
        if dim==2:
            print "does not have a degenerate point at (%d,%d)" %(x[0],x[1])
        if dim==3:
            print "does not have a degenerate point at (%d,%d,%d)" %(x[0],x[1],x[2])

def invariant2D(T,coords,info=False):
    """
    Function assumes that the tensor is given in analytical form as a
    sympy matrix. Further it only considers 2D second rank symmetric
    tensors, i.e of the form

                T = [T11 T12; T12 T22]

    The invariant of the tensor is determined :
            delta = ad - bc
    where a = d/dx(T11 - T22), d = d/dy(T12), b = d/dy(T11 - T22),
    and c = d/dx(T12). If the invariant is found to be zero, the
    tensor has two equal eigenvalues, i.e the tensor is degenerate.

    This technique is based on PhD thesis of Delmarcelle - see Del94
    """
    x0 , y0 = coords[0], coords[1]

    T11 = 0.5*T[0,0]
    T12 = 0.5*T[0,1]
    T22 = 0.5*T[1,1]

    x, y = sympy.symbols('x y')

    delta =\
    diff(T11-T22,x).evalf(subs={x:x0, y:y0})*diff(T12,y).evalf(subs={x:x0, y:y0})
    -\
```

```
    diff(T11-T22,y).evalf(subs={x:x0, y:y0})*diff(T12,x).evalf(subs={x:x0, y:y0})

    eps = 1e-12
    if abs(delta) <= eps :
        if info:
            msg(1,T,coords)
        return 1
    else:
        if info:
            msg(0,T,coords)
        return 0

def invariant3D_discriminant(T,coords,info=True):
    """
    Function assumes that the tensor is given in analytical form as a
    sympy matrix. Further it only considers 3D second rank symmetric
    tensors, i.e of the formd

                T = [T00 T01 T02;
                     T01 T11 T12;
                     T02 T12 T22]

    The discriminant of the tensor is evaluated. If it equals zero, this implies
    that the tensor has at least two equal eigenvalues, i.e the tensor is
    degenerate.

    This technique is based on the article by Zheng et. al. - see ZTP06
    """
    x0, y0, z0 = coords[0], coords[1], coords[2]
    x,y,z = sympy.symbols('x y z')

    P = T[0,0] + T[1,1] + T[2,2]
    Q = sympy.det(T[:2,:2]) + sympy.det(T[1:,1:]) + T[2,2]*T[0,0] - T[0,2]*T[0,2]
    R = sympy.det(T)

    D = Q**2*P**2 - 4*R*P**3 - 4*Q**3 - 4*Q**3 + 18*P*Q*R - 27*R**2

    D_value = D.evalf(subs={x:x0,y:y0,z:z0})
    eps = 1e-12
    if abs(D_value) <= eps :
        if info:
            msg(1,T,coords)
        return 1
    else:
        if info:
            msg(0,T,coords)
        return 0

def invariant3D_constraint_functions(T,coords,info=True):
    """
    This function is another representation of the discriminant :
    see invariant3D_discriminant().

    Function assumes that the tensor is given in analytical form as a
    sympy matrix. Further it only considers 3D second rank symmetric
    tensors, i.e of the formd

                T = [T00 T01 T02;
                     T01 T11 T12;
                     T02 T12 T22]

    The constaint function of the tensor is evaluated. If it equals zero,
    this implies that the tensor has at least two equal eigenvalues, i.e
    the tensor is degenerate.

    This technique is based on the article by Zheng et. al. - see ZTP06
    """
    x0, y0, z0 = coords[0], coords[1], coords[2]
```

```
    x,y,z = sympy.symbols('x y z')

    fx = T[0,0]*(T[1,1]**2 - T[2,2]**2) + T[0,0]*(T[0,1]**2 - T[0,2]**2)\
        + T[1,1]*(T[2,2]**2 - T[0,0]**2) + T[1,1]*(T[1,2]**2 - T[0,1]**2)\
        + T[2,2]*(T[0,0]**2 - T[1,1]**2) + T[2,2]*(T[0,2]**2 - T[1,2]**2)

    fy1 = T[1,2]*(2*(T[1,2]**2 - T[0,0]**2) - (T[0,2]**2 + T[0,1]**2)\
        + 2*(T[1,1]*T[0,0] + T[2,2]*T[0,0] - T[1,1]*T[2,2]))\
        + T[0,1]*T[0,2]*(2*T[0,0] - T[2,2] - T[1,1])

    fy2 = T[0,2]*(2*(T[0,2]**2 - T[1,1]**2) - (T[0,1]**2 + T[1,2]**2)\
        + 2*(T[2,2]*T[1,1] + T[0,0]*T[1,1] - T[2,2]*T[0,0]))\
        + T[1,2]*T[0,1]*(2*T[1,1] - T[0,0] - T[2,2])

    fy3 = T[0,1]*(2*(T[0,1]**2 - T[2,2]**2) - (T[1,2]**2 + T[0,2]**2)\
        + 2*(T[0,0]*T[2,2] + T[1,1]*T[2,2] - T[0,0]*T[1,1]))\
        + T[0,2]*T[1,2]*(2*T[2,2] - T[1,1] - T[0,0])

    fz1 = T[1,2]*(T[0,2]**2 - T[0,1]**2) + T[0,1]*T[0,2]*(T[1,1] - T[2,2])

    fz2 = T[0,2]*(T[0,1]**2 - T[1,2]**2) + T[1,2]*T[0,1]*(T[2,2] - T[0,0])

    fz3 = T[0,1]*(T[1,2]**2 - T[0,2]**2) + T[0,2]*T[1,2]*(T[0,0] - T[1,1])

    D = fx**2 + fy1**2 + fy2**2 + fy3**2 + 15*fz1**2 + 15*fz2**2 + 15*fz3**2

    D_value = D.evalf(subs={x:x0,y:y0,z:z0})
    eps = 1e-12
    if abs(D_value) <= eps :
        if info:
            msg(1,T,coords)
        return 1
    else:
        if info:
            msg(0,T,coords)
        return 0

if __name__ == "__main__":
    x,y,z = sympy.symbols('x y z')

    T1 = sympy.Matrix([[ 0.5*x**2, -x**2+y**2 ],
                      [-x**2+y**2, -0.5*x**2 - 2*x*y - 0.5*y**2]])
    x0 = 0
    y0 = 0
    coords = (x0,y0)
    invariant2D(T1,coords,info=True)

    x0 = 1
    y0 = 1
    z0 = 1
    coords = (x0,y0,z0)
    T2 = sympy.Matrix([[x**2,x*y,y**2],
                      [x*y,y**2,y*z],
                      [x**2,y*z,z**2]])
    discrim = invariant3D_discriminant(T2,coords,info=True)
    constrain = invariant3D_constraint_functions(T2,coords,info=True)

    if discrim == constrain:
        print "Both functions give same result!"
```

C.3 Find the Metric g_{ij}

find_metric.py

```
import sympy as sym

def curve_to_metric(ds2,dim,diff_=None):
```

```
    if dim < 2:
        raise ValueError("The metric is implemented for at least dim = 2.")
    ds2 = str(ds2)
    ds2 = ds2.replace(' ','')
    differentials = []
    if diff_ is None:
        if dim >= 2:
            differentials = ['du','dv']
        if dim >= 3:
            differentials.append('dw')
        if dim == 4:
            differentials.append('dt')
    else:
        M = sym.Matrix(diff_)
        for i in range(0,M.shape[0]):
            n = str(M[i,i]).find('*')
            diff = str(M[i,i])[0:n]
            differentials.append(diff)

    for diff in differentials:
        ds2 = ds2.replace(diff+'**2',diff+'*'+diff)

    def split_elements(expr,tmp_list,side='right'):
        new_list = []
        if type(tmp_list) is list:
            for term in tmp_list:
                split_terms = term.split(expr)
                if type(split_terms) is list:
                    for sterms in split_terms:
                        new_list.append(sterms)
                else:
                    new_list.append(split_terms)
        else:
            split_terms = tmp_list.split(expr) # split at found expression
            if type(split_terms) is list:
                for sterms in split_terms:
                    new_list.append(sterms)
            else:
                new_list.append(split_terms)

        lost = expr[:-1]
        sign = '-' # special case to be handeled
        if side=='left':
            lost = expr[1:]
        for i in range(len(new_list)):
            term = new_list[i]
            if side=='left':
                first = 0
                if term[first] == '*':
                    new_list[i] = lost+new_list[i]
            else:
                last = len(term) - 1
                if term[last] == '*':
                    new_list[i] = new_list[i]+lost
                    if expr[-1] == sign: # if expression contains '-' at end
                        # for next in list add '-'
                        new_list[i+1] = '-'+new_list[i+1]
        return new_list

    n = ds2
    L='left'
    R='right'
    p = '+'
    m = '-'
    for diff in differentials: # for each differential : dx_i = du,dv,dw,dt
        expr = diff+p # 'dx_i+'
        n = split_elements(expr,n,R)
        expr = diff+m # 'dx_i-'
```

```
        n = split_elements(expr,n,R)
        expr = p+diff # '+dx_i'
        n = split_elements(expr,n,L)
        expr = m+diff # '-dx_i'
        n = split_elements(expr,n,L)

    # define the matrix structure of the metric components for mapping the
    if dim == 2: # curve elements to their corresponding location
        if diff_ is None:
            diff = [['du*du','du*dv'],
                    ['dv*du','dv*dv']]
        else:
            diff = diff_
    if dim == 3:
        if diff_ is None:
            diff = [['du*du','du*dv','du*dw'],
                     ['dv*du','dv*dv','dv*dw'],
                     ['dw*du','dw*dv','dw*dw']]
        else:
            diff = diff_
    if dim == 4:
        if diff_ is None:
            diff = [['du*du','du*dv','du*dw','dt*du'],
                     ['dv*du','dv*dv','dv*dw','dv*dt'],
                     ['dw*du','dv*dw','dw*dw','dw*dt'],
                     ['dt*du','dt*dv','dt*dw','dt*dt']]
        else:
            diff = diff_
    # add the elements in g without the above differentials
    elements = n
    g = [['0' for _ in range(dim)] for _ in range(dim)]
    for element in elements:
        for i in range(dim):
            for j in range(dim):
                if (element.find(diff[i][j]) != -1):
                    if (element.find('*' + diff[i][j]) != -1):
                        g[i][j] = element.replace('*'+ diff[i][j],'')
                    else:
                        g[i][j] = element.replace(diff[i][j],'1')
                    g[j][i] = g[i][j]
    from sympy import Matrix, sin, cos, exp, log, cosh, sinh, sqrt, tan, tanh
    from sympy.abc import u,v,w,t
    return Matrix(g)


def metric(coord1,coord2,form="simplified",write_to_file=False):
    """
    Calculates the metric for the coordinate transformation
    between cartesian coordinates to another orthogonal
    coordinate system.
    """
    from sympy import diff
    x,y = coord1[0], coord1[1]
    u,v = coord2[0], coord2[1]
    dim = len(coord1)
    if len(coord2) != dim:
        import sys
        sys.exit("Coordinate systems must have same dimensions.")
    if dim >= 3:
        z = coord1[2]
        w = coord2[2]
    if dim == 4:
        t1 = coord1[3]
        t2 = coord2[3]
    dxdu = diff(x,u)
    dxdv = diff(x,v)
    dydu = diff(y,u)
    dydv = diff(y,v)
```

```
    if dim >= 3:
        dxdw = diff(x,w)
        dydw = diff(y,w)
        dzdu = diff(z,u)
        dzdv = diff(z,v)
        dzdw = diff(z,w)
    if dim == 4:
        dxdt = diff(x,t2)
        dydt = diff(y,t2)
        dzdt = diff(z,t2)
        dtdu = diff(t1,u)
        dtdv = diff(t1,v)
        dtdw = diff(t1,w)
        dtdt = diff(t1,t2)


    import numpy as np
    from sympy import Matrix
    g = Matrix(np.zeros([dim,dim]))
    g[0,0] = dxdu*dxdu + dydu*dydu
    g[0,1] = dxdu*dxdv + dydu*dydv
    g[1,1] = dxdv*dxdv + dydv*dydv
    g[1,0] = g[0,1]
    if dim >= 3:
        g[0,0] += dzdu*dzdu
        g[0,1] += dzdu*dzdv; g[1,0] = g[0,1]
        g[0,2] = dxdu*dxdw + dydu*dydw + dzdu*dzdw; g[2,0] = g[0,2]
        g[1,1] += dzdv*dzdv
        g[1,2] = dxdv*dxdw + dydv*dydw + dzdv*dzdw; g[2,1] = g[1,2]
        g[2,2] = dxdw*dxdw + dydw*dydw + dzdw*dzdw
    if dim == 4:
        g[0,0] += dtdu*dtdu
        g[0,1] += dtdu*dtdv; g[1,0] = g[0,1]
        g[0,2] += dtdu*dtdw; g[2,0] = g[0,2]
        g[0,3] = dxdu*dxdt + dydu*dydt + dzdu*dzdt + dtdu*dtdt; g[3,0] = g[0,3]
        g[1,1] += dtdv*dtdv
        g[1,2] += dtdv*dtdw; g[2,1] = g[1,2]
        g[1,3] = dxdv*dxdt + dydv*dydt + dzdv*dzdt + dtdv*dtdt; g[3,1] = g[1,3]
        g[2,2] += dtdw*dtdw
        g[2,3] = dxdw*dxdt + dydw*dydt + dzdw*dzdt + dtdw*dtdt; g[3,2] = g[2,3]
        g[3,3] = dxdt*dtdt + dydt*dtdt + dzdt*dtdt + dtdt*dtdt

    if form=="simplified":
        def symplify_expr(expr):
            new_expr = sym.trigsimp(expr)
            new_expr = sym.simplify(new_expr)
            return new_expr
        print "Performing simplification on the metric. This may take some time ...."
        for i in range(0,dim):
            for j in range(0,dim):
                g[i,j] = symplify_expr(g[i,j])

    if write_to_file:
        f = open("metric.txt","w")
        f.write(str(g))
        f.close()
    return g

def toroidal_coordinates(form="simplified"):
    from sympy.abc import u, v, w, a
    from sympy import sin,cos,sinh,cosh

    x = (a*sinh(u)*cos(w))/(cosh(u) - cos(v))
    y = (a*sinh(u)*sin(w))/(cosh(u) - cos(v))
    z = (a*sin(v))/(cosh(u) - cos(v))
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
```

```python
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def cylindrical_coordinates(form="simplified"):
    from sympy.abc import u, v, w, x, y, z
    from sympy import sin,cos

    x = u*cos(v)
    y = u*sin(v)
    z = w
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def spherical_coordinates(form="simplified"):
    from sympy.abc import u, v, w, x, y, z
    from sympy import sin,cos

    x = u*sin(v)*cos(w)
    y = u*sin(v)*sin(w)
    z = u*cos(v)
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def inverse_prolate_spheroidal_coordinates(form="usimp",write_to_file=True):
    from sympy.abc import u, v, w, a
    from sympy import sin,cos,sinh,cosh

    x = (a*sinh(u)*sin(v)*cos(w))/(cosh(u)**2 - sin(v)**2)
    y = (a*sinh(u)*sin(v)*sin(w))/(cosh(u)**2 - sin(v)**2)
    z = (a*cosh(u)*cos(v))/(cosh(u)**2 - sin(v)**2)
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form,write_to_file)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def cylindrical_catenoid_coordinates(form='simplified'):
    from sympy.abc import u, v, w
    from sympy import sin,cos
    x = cos(u) - v*sin(u)
    y = sin(u) + v*cos(u)
    z = v
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    g = g[:2,:2] # 2-dimensional
    diff_form = [['du*du','du*dv'],
                 ['dv*du','dv*dv']]
    return g, diff_form

def egg_carton_coordinates(form='simplified'):
    from sympy.abc import u, v, w
    from sympy import sin,cos
    x = u
```

```
    y = v
    z = sin(u)*cos(v)
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    g = g[:2,:2] # 2-dimensional
    diff_form = [['du*du','du*dv'],
                 ['dv*du','dv*dv']]
    return g, diff_form

def analytical(k_value=0,form="simplified"): # k=0 gives flat space
    from sympy import sin
    from sympy.abc import u,v,k
    ds2 = '(1/(1 - k*u**2))*du**2 + u**2*dv**2 + u**2*sin(v)**2*dw**2'
    g = curve_to_metric(ds2,3)
    g = g.subs(k,k_value)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def kerr_metric(): #in polar coordinates u,v,w, and t
    from sympy import symbols, simplify, cos, sin
    from sympy.abc import G,M,l,u,v,w #,c,J
    # from wikipedia :
    """
    ds2 = '(1 - us*u/p)*c**2*dt**2 - (p/l)*du**2 - p*dv**2 -\
          (u**2 + a**2 + (us*u*a**2/p**2)*sin(v)**2)*sin(v)**2*dw**2\
          + (2*us*u*a*sin(v)**2/p)*c*dt*dw'
    g = curve_to_metric(ds2,dim=4)

    us,p,a,l = symbols('us,p,a,l')
    g = g.subs({p:u**2 + a**2*cos(v)})
    g = g.subs({l:u**2 - us*u + a**2})
    g = g.subs({us:2*G*M/c**2})
    g = g.subs({a:J/(M*c)})
    """
    # from Thomas A. Moore (if a=0 ds2 reduces to Schwarzchild solution)
    ds2 = '-(1 - us*u/p)*dt**2 + (p/l)*du**2 + p*dv**2 \
           + (u**2 + a**2 + (us*u*a**2*sin(v)**2/p**2))*sin(v)**2*dw**2\
           - (2*us*u*a*sin(v)**2/p)*dt*dw'
    g = curve_to_metric(ds2,dim=4)
    us,p,a,l = symbols('us,p,a,l')
    g = g.subs({p:u**2 + a**2*cos(v)})
    g = g.subs({l:u**2 - us*u + a**2})
    g = g.subs({us:2*G*M})
    print "Performing simplification on the metric. This may take some time ...."
    g = simplify(g)
    diff_form = [['du*du','du*dv','du*dw','dt*du'],
                 ['dv*du','dv*dv','dv*dw','dv*dt'],
                 ['dw*du','dv*dw','dw*dw','dw*dt'],
                 ['dt*du','dt*dv','dt*dw','dt*dt']]
    return g, diff_form

def kerr_3D_metric_time_independent(): # unphysical ?
    from sympy import symbols, simplify, cos, sin
    from sympy.abc import G,M,l,u,v,w
    ds2 = '(p/l)*du**2 + p*dv**2 \
           + (u**2 + a**2 + (us*u*a**2*sin(v)**2/p**2))*sin(v)**2*dw**2'
    g = curve_to_metric(ds2,dim=3)
    us,p,a,l = symbols('us,p,a,l')
    g = g.subs({p:u**2 + a**2*cos(v)})
    g = g.subs({l:u**2 - us*u + a**2})
    g = g.subs({us:2*G*M})
    print "Performing simplification on the metric. This may take some time ...."
    g = simplify(g)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
```

```
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def kerr_3D_metric(): # one space component dropped : phi
    from sympy import symbols, simplify, cos, sin
    from sympy.abc import G,M,l,u,v,t
    ds2 = '-(1 - us*u/p)*dw**2 + (p/l)*du**2 + p*dv**2'
    g = curve_to_metric(ds2,dim=3)
    us,p,a,l = symbols('us,p,a,l')
    g = g.subs({p:u**2 + a**2*cos(v)})
    g = g.subs({l:u**2 - us*u + a**2})
    g = g.subs({us:2*G*M})
    print "Performing simplification on the metric. This may take some time ...."
    g = simplify(g)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

def torus_metric(a=1,c=2,form='simplified'):
    from sympy.abc import u, v, w
    from sympy import sin,cos
    x = (c + a*cos(u))*cos(v)
    y = (c + a*cos(u))*sin(v)
    z = sin(u)
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    g = g[:2,:2]
    diff_form = [['du*du','du*dv'],
                 ['dv*du','dv*dv']]
    return g, diff_form

def flat_sphere():
    ds2 = 'dv**2 + sin(v)**2*dw**2'
    diff_form = [['dv*dv','dv*dw'],['dw*dv','dw*dw']]
    g = curve_to_metric(ds2,dim=2,diff_=diff_form)
    return g, diff_form

def mobius_strip(form='simplified'):
    from sympy.abc import u, v, w
    from sympy import sin,cos
    x = (1 + cos(u/2)*v/2)*cos(u)
    y = (1 + cos(u/2)*v/2)*sin(u)
    z = sin(u/2)*v/2
    coord1 = (x,y,z)
    coord2 = (u,v,w)
    g = metric(coord1,coord2,form)
    g = g[:2,:2]
    diff_form = [['du*du','du*dv'],['dv*du','dv*dv']]
    return g, diff_form

if __name__ == "__main__":
    g1,diff_form = toroidal_coordinates()
    print "The toroidal metric"
    print g1
    print 'with the corresponding differentials'
    print diff_form

    """
    g2,diff_form = cylindrical_coordinates()
    print "\nCylindrical"
    print g2

    g3, diff_form = spherical_coordinates()
    print "\nSpherical"
    print g3

```

```
    g4, diff_form = inverse_prolate_spheroidal_coordinates("usimp",1)
    print "\nInverse prolate spheroidal coordinates - without simplified form"
    print g4
    """
```

C.4 Riemann Curvature, Ricci tensor, Scalar curvature

tensor.py

```
from __future__ import division
import sympy as sympy

class Riemann:
    """
    Used for defining a Riemann curvature tensor or Ricci tensor
    for a given metric between cartesian coordinates and another
    orthognal coordinate system.
    """
    def __init__(self, g, dim, sys_title="coordinate_system", user_coord = None,
                 flat_diff = None):
        """
        Contructor __init__ initializes the object for a given
        metric g (symbolic matrix). The metric must be defined
        with sympy variables u and v for the orthoganl basis in
        the Cartesian coordinate system.

        g : metric defined as nxn Sympy.Matrix object
        sys_titles : descriptive information about the coordinate system
                     besides Cartesian coordinates
        dim : R^2, R^3, or R^4
        user_coord : User supplies their own set of coordinate symbols
        flat_diff : Matrix with differentials if g is a flat metric
        """
        from sympy.diffgeom import Manifold, Patch
        self.dim = dim
        self.g = g
        if flat_diff is not None:
            self._set_flat_coordinates(sys_title,flat_diff)
        elif user_coord is None:
            self._set_coordinates(sys_title)
        else:
            self._set_user_coordinates(sys_title,user_coord)
        self.metric = self._metric_to_twoform(g)

    def _set_flat_coordinates(self,sys_title,flat_diff):
        from sympy.diffgeom import CoordSystem, Manifold, Patch
        manifold = Manifold("M",self.dim)
        patch = Patch("P",manifold)
        flat_diff = sympy.Matrix(flat_diff)
        N = flat_diff.shape[0]
        coords = []
        for i in range(0,N):
            n = str(flat_diff[i,i]).find('*')
            coord_i = str(flat_diff[i,i])[1:n]
            coords.append(coord_i)
        if self.dim==4:
            system = CoordSystem(sys_title, patch, [str(coords[0]),str(coords[1]),\
                                                    str(coords[2]),str(coords[3])])
            u, v, w, t = system.coord_functions()
            self.w = w
            self.t = t

        if self.dim==3:
            system = CoordSystem(sys_title, patch, [str(coords[0]),str(coords[1]),\
                                                   str(coords[2])])
            u, v, w = system.coord_functions()
            self.w = w
```

```

        if self.dim==2:
            system = CoordSystem(sys_title, patch, [str(coords[0]),str(coords[1])])
            u, v = system.coord_functions()

        self.u, self.v = u, v
        self.system = system

    def _set_user_coordinates(self,sys_title,user_coord):
        from sympy.diffgeom import CoordSystem, Manifold, Patch
        manifold = Manifold("M",self.dim)
        patch = Patch("P",manifold)
        if self.dim==4:
            system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1]),\
                                                 str(user_coord[2]),str(user_coord[3])])
            u, v, w, t = system.coord_functions()
            self.w = w
            self.t = t

        if self.dim==3:
            system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1]),
                                                 str(user_coord[2])])
            u, v, w = system.coord_functions()
            self.w = w

        if self.dim==2:
            system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1])])
            u, v = system.coord_functions()

        self.u, self.v = u, v
        self.system = system

    def _set_coordinates(self,sys_title):
        from sympy.diffgeom import CoordSystem, Manifold, Patch
        manifold = Manifold("M",self.dim)
        patch = Patch("P",manifold)
        if self.dim==4:
            system = CoordSystem(sys_title, patch, ["u", "v", "w","t"])
            u, v, w, t = system.coord_functions()
            self.w = w
            self.t = t

        if self.dim==3:
            system = CoordSystem(sys_title, patch, ["u", "v", "w"])
            u, v, w = system.coord_functions()
            self.w = w

        if self.dim==2:
            system = CoordSystem(sys_title, patch, ["u", "v"])
            u, v = system.coord_functions()

        self.u, self.v = u, v
        self.system = system

    def _metric_to_twoform(self,g):
        dim = self.dim
        system = self.system
        diff_forms = system.base_oneforms()
        u_, v_ = self.u, self.v
        u = u_
        v = v_
        if dim >= 3:
            w_ = self.w
            w = w_
        if dim == 4:
            t_ = self.t
            t = t_

```

```python
        from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
        import sympy.abc as abc
        self._abc = abc
        self._symbols = ['*','/','(',')',"'",'"']
        self._letters = []
        g_ = sympy.Matrix(dim*[dim*[0]])
        # re-evaluate the metric for (u,v,w,t if 4D) which are Basescalar objects
        for i in range(dim):
            for j in range(dim):
                expr = str(g[i,j])
                self._try_expr(expr) # evaluate expr in a safe environment
                for letter in self._letters:
                    exec('from sympy.abc import %s'%letter)
                g_[i,j] = eval(expr) # this will now work for any variables defined in sympy.abc
                g_[i,j] = g_[i,j].subs(u,u_)
                g_[i,j] = g_[i,j].subs(v,v_)
                if dim >= 3:
                    g_[i,j] = g_[i,j].subs(w,w_)
                if dim == 4:
                    g_[i,j] = g_[i,j].subs(t,t_)
        from sympy.diffgeom import TensorProduct
        metric_diff_form = sum([TensorProduct(di, dj)*g_[i, j]
                                for i, di in enumerate(diff_forms)
                                for j, dj in enumerate(diff_forms)])
        return metric_diff_form

    def _try_expr(self,expr):
        """
        This is a help function used initially to evaluate the user-defined metric
        elements as a sympy expression : expr. The purpose of this method is to
        prevent the namespace of the user from being polluted by the command
        'from sympy.abc import *'.

        expr : a string object to be evaluated as a sympy expression
        """
        from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
        letters = self._letters
        abc = self._abc
        try:
            for letter in letters:
                exec('from sympy.abc import %s'%letter) # re-execute after finding each unknown variable
            l_ = expr.count('(')
            r_ = expr.count(')')
            if l_ == r_:
                eval(expr)
            elif l_ < r_:
                eval((r_-l_)*'('+expr)
        except NameError as err:
            msg = str(err)
            pos = msg.find("'")
            letter = msg[pos+1]
            pos = pos +1
            found = False
            symbols = self._symbols
            while (pos+1 < len(msg)) and (not found):
                more = msg[pos+1]
                for symb in symbols:
                    if more==symb or more.isdigit():
                        found = True
                        break
                if found is False:
                    letter = letter+more
                    pos = pos + 1
            for alphabet in abc.__dict__:
                if letter == alphabet:
                    letters.append(alphabet)
                    self._try_expr(expr[expr.find(alphabet):]) # search for the next unknown variable

```

```python
    def _tuple_to_list(self,t):
        """
        Recoursively turn a tuple to a list.
        """
        return list(map(self._tuple_to_list, t)) if isinstance(t, (list, tuple)) else t

    def _symplify_expr(self,expr): # this is a costly stage for complex expressions
            """
            Perform simplification of the provided expression.
            Method returns a SymPy expression.
            """
            expr = sympy.trigsimp(expr)
            expr = sympy.simplify(expr)
            return expr

    def metric_to_Christoffel_1st(self):
        from sympy.diffgeom import metric_to_Christoffel_1st
        return metric_to_Christoffel_1st(self.metric)

    def metric_to_Christoffel_2nd(self):
        from sympy.diffgeom import metric_to_Christoffel_2nd
        return metric_to_Christoffel_2nd(self.metric)

    def find_Christoffel_tensor(self,form="simplified"):
        """
        Method determines the Riemann-Christoffel tensor
        for a given metric(which must be in two-form).

        form : default value - "simplified"
        If desired, a simplified form is returned.

        The returned value is a SymPy Matrix.
        """
        from sympy.diffgeom import metric_to_Riemann_components
        metric = self.metric
        R = metric_to_Riemann_components(metric)
        simpR = self._tuple_to_list(R)
        dim = self.dim
        if form=="simplified":
            print 'Performing simplifications on each component....'
            for m in range(dim):
                for i in range(dim):
                    for j in range(dim):
                        for k in range(dim):
                            expr = str(R[m][i][j][k])
                            expr = self._symplify_expr(expr)
                            simpR[m][i][j][k] = expr
        self.Christoffel = sympy.Matrix(simpR)
        return self.Christoffel

    def find_Ricci_tensor(self,form="simplified"):
        """
        Method determines the Ricci curvature tensor for
        a given metric(which must be in two-form).

        form : default value - "simplified"
        If desired, a simplified form is returned.

        The returned value is a SymPy Matrix.
        """
        from sympy.diffgeom import metric_to_Ricci_components
        metric = self.metric
        RR = metric_to_Ricci_components(metric)
        simpRR = self._tuple_to_list(RR)
        dim = self.dim
        if form=="simplified":
            print 'Performing simplifications on each component....'
            for m in range(dim):
```

```
                for i in range(dim):
                    expr = str(RR[m][i])
                    expr = self._symplify_expr(expr)
                    simpRR[m][i] = expr
        self.Ricci = sympy.Matrix(simpRR)
        return self.Ricci

    def find_scalar_curvature(self):
        """
        Method performs scalar contraction on the Ricci tensor.
        """
        try:
            Ricci = self.Ricci
        except AttributeError:
            print "Ricci tensor must be determined first."
            return None
        g = self.g
        g_inv = self.g.inv()
        scalar_curv = sympy.simplify(g_inv*Ricci)
        scalar_curv = sympy.trace(scalar_curv)
        self.scalar_curv = scalar_curv
        return self.scalar_curv


if __name__ == "__main__":
    import find_metric
    k = -1
    g,diff_form = find_metric.analytical(k) # k=0 gives flat space
    R = Riemann(g,dim=3,sys_title="analytical")
    print R.metric
    from sympy import srepr
    print srepr(R.system)
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    scalarRR = R.find_scalar_curvature()

    print "\nThe analytical curve element has the following metric for k=%.1f"%k
    print g
    print "\nThe Ricci tensor is given as"
    print RR
    print "\nand the scalar curvature is"
    print scalarRR

    """
    from sympy.abc import r,theta, phi, u,v
    g,diff_form = find_metric.flat_sphere()
    diff = [['dv*dv','dv*dw'],['dw*dv','dw*dw']]
    R = Riemann(g, dim=2, sys_title="flat_sphere",\
                flat_metric = True, flat_diff = diff)
    C = R.metric_to_Christoffel_2nd(R.metric)
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    scalarRR = R.find_scalar_curvature()

    print "\nThe 2D sphere has the following metric"
    print g
    print "\nThe Christoffel tensor is given as"
    for m in range(dim):
        for i in range(dim):
            print RC[m,i]
    print "\nThe Ricci tensor is given as"
    print RR
    print "\nand the scalar curvature is"
    print scalarRR


    g,diff_form = find_metric.toroidal_coordinates()
    R = Riemann(g,dim=3,sys_title="toroidal")
```

```
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    print RC,"\n",RR

    g,diff_form = find_metric.spherical_coordinates()
    R = Riemann(g,dim=3,sys_title="spherical")
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    print RC,"\n",RR

    g,diff_form = find_metric.cylindrical_coordinates()
    R = Riemann(g=g,dim=3,sys_title="cylindrical")
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    print RC,"\n",RR

    # Warning : This takes very long time (just to find g)!
    g,diff_form = find_metric.inverse_prolate_spheroidal_coordinates()
    R = Riemann(g,dim=3,sys_title="inv_prolate_sphere")
    RC = R.find_Christoffel_tensor()
    RR = R.find_Ricci_tensor()
    print RC,"\n",RR
    """
```

C.5 Geodesic Differential Equations Solver

gde.py

```
import numpy as np
import scipy.integrate as sc
import sympy as sym

def f3D(y,s,*args):
    C,u,v,w = args
    y0 = y[0] # u
    y1 = y[1] # u'
    y2 = y[2] # v
    y3 = y[3] # v'
    y4 = y[4] # w
    y5 = y[5] # w'
    C = C.subs({u:y0,v:y2,w:y4})

    dy = np.zeros_like(y)
    dy[0] = y1
    dy[2] = y3
    dy[4] = y5
    dy[1] = -C[0,0][0]*dy[0]**2\
          -2*C[0,0][1]*dy[0]*dy[2]\
          -2*C[0,0][2]*dy[0]*dy[4]\
          -2*C[0,1][2]*dy[2]*dy[4]\
            -C[0,1][1]*dy[2]**2\
            -C[0,2][2]*dy[4]**2
    dy[3] = -C[1,0][0]*dy[0]**2\
          -2*C[1,0][1]*dy[0]*dy[2]\
          -2*C[1,0][2]*dy[0]*dy[4]\
          -2*C[1,1][2]*dy[2]*dy[4]\
            -C[1,1][1]*dy[2]**2\
            -C[1,2][2]*dy[4]**2
    dy[5] = -C[2,0][0]*dy[0]**2\
          -2*C[2,0][1]*dy[0]*dy[2]\
          -2*C[2,0][2]*dy[0]*dy[4]\
          -2*C[2,1][2]*dy[2]*dy[4]\
            -C[2,1][1]*dy[2]**2\
            -C[2,2][2]*dy[4]**2
    return dy

def f(y,s,*args):
```

```
    """
    The geodesic differential equations are solved.
    Described as a system of first order differential-
    equations :

    y0 = u
    y1 = u'
    y2 = v
    y3 = v'

    dy0 = y1
    dy1 = u''
    dy2 = y2
    dy3 = v''

    Input :
    C is the Christoffel symbol of second kind
    u and v are symbolic expressions.

    Output :
    dy = [dy0,dy1,dy2,dy3]
    """
    C,u,v = args
    y0 = y[0] # u
    y1 = y[1] # u'
    y2 = y[2] # v
    y3 = y[3] # v'
    dy = np.zeros_like(y)
    dy[0] = y1
    dy[2] = y3

    C = C.subs({u:y0,v:y2})
    dy[1] = -C[0,0][0]*dy[0]**2\
           -2*C[0,0][1]*dy[0]*dy[2]\
             -C[0,1][1]*dy[2]**2
    dy[3] = -C[1,0][0]*dy[0]**2\
           -2*C[1,0][1]*dy[0]*dy[2]\
             -C[1,1][1]*dy[2]**2
    return dy

def solve(C,u0,s0,s1,ds,solver=None):
    from sympy.abc import u,v
    global f
    if len(u0) == 6: # 3D problem
        from sympy.abc import w
        args = (C,u,v,w)
        f = f3D
    else:
        args = (C,u,v)

    if solver == None: # use lsoda from scipy.integrate.odeint
        s = np.arange(s0,s1+ds,ds)
        print 'Running solver ...'
        return sc.odeint(f,u0,s,args=args)
    else: # use any other solver from scipy.integrate.ode
        # vode,zvode,lsoda,dopri5,dop853
        r = sc.ode(lambda t,x,args: f(x,t,*args)).set_integrator(solver)
        r.set_f_params(args)
        r.set_initial_value(u0)
        y = []
        print 'Running solver ...'
        while r.successful() and r.t <= s1:
            r.integrate(r.t + ds)
            y.append(r.y)
        return np.array(y)

def two_points(p1,p2,s0,s1,ds,C,tol=1e-6,surface=None):
    """
```

```
    The function attempts to find the geodesic between two points p1 and p2.
    """
    p1 = np.array(p1)
    p2 = np.array(p2)
    if (np.fabs(p1-p2) <= tol).all() == 1:
        raise ValueError('Point 1 and point 2 are the same point : (%.1f,%.1f)'%(p2[0],p2[1]))
    found = False
    X_ = []
    u_ = 4*[0]; u_[0] = p1[0]; u_[2] = p1[1]
    du = np.arange(-.2,.2,ds)
    N = du.shape[0]
    i = 0
    while (i < N) and (not found):
        u_[1] = du[i]
        j = 0
        while (j < N) and (not found):
            u_[3] = du[j]
            print 'Testing initial conditions :'
            print u_
            X = solve(C,u_,s0,s1,ds)
            u__ = np.where(np.fabs(X[:,0]-p2[0]) <= tol)[0]
            v__ = np.where(np.fabs(X[:,2]-p2[1]) <= tol)[0]
            if (u__ == v__).any() == True:
                found = True
                X_ = X
                print 'Following initial conditions connect the two provided points '
                print '(%.6f,%.6f)'%(u_[0],u_[2]), ', (%.6f,%.6f)'%(p2[0],p2[1])
                print "u' = %f , v' = %f"%(u_[1],u_[3])
            j = j + 1
        i = i + 1
    if (len(X_) > 0) and (surface is not None):
        print 'Plotting the geodesics for provided surface...',surface
        if surface == 'catenoid':
            display_catenoid(u_,s0,s1,ds,show=True)
        elif surface == 'torus':
            display_torus(u_,s0,s1,ds,show=True)
        elif surface == 'sphere':
            display_sphere(u_,s0,s1,ds,show=True)
        elif surface == 'egg_carton':
            display_egg_carton(u_,s0,s1,ds,show=True)
    return X

def Christoffel_2nd(g=None,metric=None): # either g is supplied as arugment or the two-form
    from sympy.abc import u,v
    from sympy.diffgeom import metric_to_Christoffel_2nd
    from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
    if metric is None: # if metric is not specified as two_form
        import tensor as t
        R = t.Riemann(g,g.shape[0])
        metric = R.metric
    C = sym.Matrix(eval(str(metric_to_Christoffel_2nd(metric))))
    return C

def catenoid():
    import find_metric
    g = find_metric.cylindrical_catenoid_coordinates()
    C = Christoffel_2nd(g)
    return C

def torus(a=1,c=2):
    import find_metric
    g = find_metric.torus_metric(a,c)
    C = Christoffel_2nd(g)
    return C

def toroid(u=1,v=None,a=1):
    import find_metric as fm
    g, diff = fm.toroidal_coordinates()
```

```
    if v is None:
        g = g.subs('u',u)[:2,:2]
    else:
        g = g.subs('v',v)[1:,1:]
    g = g.subs('a',a)

    import tensor as t
    R = t.Riemann(g,dim=2,sys_title='toroid')
    C = Christoffel_2nd(metric=R.metric)
    return C


def egg_carton():
    import tensor as t
    import find_metric as fm
    g,diff = find_metric.egg_carton_metric()
    R = t.Riemann(g,dim=2,sys_title='egg_carton',flat_diff=diff)
    """
    # this works :
    from sympy.abc import u,v
    u_,v_ = R.system.coord_functions()
    du,dv = R.system.base_oneforms()
    metric = R.metric.subs({u:u_,v:v_,'dv':dv,'du':du})
    """
    C = Christoffel_2nd(metric=R.metric)
    return C

def flat_kerr(a=0,G=1,M=0.5):
    import find_metric as fm
    from sympy.diffgeom import CoordSystem, Manifold, Patch, TensorProduct

    manifold = Manifold("M",3)
    patch = Patch("P",manifold)
    kerr = CoordSystem("kerr", patch, ["u","v","w"])
    u,v,w = kerr.coord_functions()
    du,dv,dw = kerr.base_oneforms()

    g11 = (a**2*sym.cos(v) + u**2)/(-2*G*M*u + a**2 + u**2)
    g22 = a**2*sym.cos(v) + u**2
    g33 = -(1 - 2*G*M*u/(u**2 + a**2*sym.cos(v)))
    # time independent : unphysical ?
    #g33 = 2*G*M*a**2*sym.sin(v)**4*u/(a**2*sym.cos(v) + u**2)**2 + a**2*sym.sin(v)**2 + sym.sin(v)**2*u**2
    metric = g11*TensorProduct(du, du) + g22*TensorProduct(dv, dv) + g33*TensorProduct(dw, dw)
    C = Christoffel_2nd(metric=metric)
    return C

def flat_sphere():
    import find_metric as fm
    import tensor as t
    g,diff = find_metric.flat_sphere()
    R = t.Riemann(g,dim=2,sys_title='flat_sphere',flat_diff=diff)
    C = Christoffel_2nd(metric=R.metric)
    return C

def sphere():
    from sympy.abc import u,v
    from sympy import tan, cos ,sin
    """
    return flat_sphere() # in correct entries in Christoffel symbol of 2nd kind
    """
    return sym.Matrix([[(0,-tan(v)), (0, 0)],[(sin(v)*cos(v), 0), (0, 0)]])

def mobius_strip():
    import find_metric as fm
    import tensor as t
    g,diff = fm.mobius_strip()
    R = t.Riemann(g,dim=2,sys_title='mobius_strip',flat_diff = diff)
```

```
    #metric=R.metric
    from sympy.diffgeom import TensorProduct, Manifold, Patch, CoordSystem
    manifold = Manifold("M",2)
    patch = Patch("P",manifold)
    system = CoordSystem('mobius_strip', patch, ["u", "v"])
    u, v = system.coord_functions()
    du,dv = system.base_oneforms()
    from sympy import cos
    metric = (cos(u/2)**2*v**2/4 + cos(u/2)*v + v**2/16 + 1)*TensorProduct(du, du) + 0.25*TensorProduct(dv, dv)
    C = Christoffel_2nd(metric=metric)
    return C

def display_mobius_strip(u0,s0,s1,ds,solver=None,show=False):
    C = mobius_strip() # Find the Christoffel tensor for mobius strip
    X = solve(C,u0,s0,s1,ds,solver)

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    u,v = plt.meshgrid(np.linspace(-2*np.pi,np.pi,250),np.linspace(-np.pi,np.pi,250))
    x = (1 + np.cos(u/2.)*v/2.)*np.cos(u)
    y = (1 + np.cos(u/2.)*v/2.)*np.sin(u)
    z = np.sin(u/2.)*v/2.

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(elev=10, azim=81)
    # use transparent colormap
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = -.5*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    plt.hold('on')

    # plot the parametrized data on to the catenoid
    u,v = X[:,0], X[:,2]
    x = (1 + np.cos(u/2.)*v/2.)*np.cos(u)
    y = (1 + np.cos(u/2.)*v/2.)*np.sin(u)
    z = np.sin(u/2.)*v/2.

    ax.plot(x,y,z,'--r')
    s0_ = s0/np.pi
    s1_ = s1/np.pi
    fig.suptitle("$s\in[%1.f,%1.f\pi]$ , $u = %.1f$ , $u' = %.1f$, $v = %.1f$ , $v' = %.1f$"%(s0,s1_,u0[0],u0[1],u0[2],u0[3]))
    if show == True:
        plt.show()
    return X,plt

def display_catenoid(u0,s0,s1,ds,solver=None,show=False):
    C = catenoid() # Find the Christoffel tensor for cylindrical catenoid
    X = solve(C,u0,s0,s1,ds,solver)

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    N = X[:,0].shape[0]
    u,v = plt.meshgrid(np.linspace(-np.pi,np.pi,150),np.linspace(-np.pi,np.pi,150))
    x = np.cos(u) - v*np.sin(u)
    y = np.sin(u) + v*np.cos(u)
    z = v

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
```

```
    ax.view_init(elev=20, azim=-163)
    # use transparent colormap
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = -.5*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

    # plot the parametrized data on to the catenoid
    u,v = X[:,0], X[:,2]
    x = np.cos(u) - v*np.sin(u)
    y = np.sin(u) + v*np.cos(u)
    z = v

    ax.plot(x,y,z,'--r')
    s0_ = s0/np.pi
    s1_ = s1/np.pi
    fig.suptitle("$s\in[%.1f\pi,%.1f\pi]$ , $u' = %.1f$ , $v' = %.2f$"%(s0_,s1_,u0[1],u0[3]))
    if show == True:
        plt.show()
    return X,plt

def display_sphere(u0,s0,s1,ds,solver=None,metric=None,show=False):
    if metric == 'flat':
        C = flat_sphere()
        if u0[0] == 0 or u0[2] == 0:
            print 'Division by zero may occur for provided values of u(s0) and v(s0)'
    else:
        C = sphere()
    X = solve(C,u0,s0,s1,ds,solver)
    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
    x = np.cos(u)*np.cos(v)
    y = np.sin(u)*np.cos(v)
    z = np.sin(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    if metric == 'flat':
        ax.view_init(elev=90., azim=0)
    else:
        ax.view_init(elev=0., azim=13)
    ax.plot_surface(x,y,z,linewidth=0,cmap='Pastel1')
    plt.hold('on')
    # plot the parametrized data on to the sphere
    u,v = X[:,0], X[:,2]
    x = np.cos(u)*np.cos(v)
    y = np.sin(u)*np.cos(v)
    z = np.sin(v)

    ax.plot(x,y,z,'--r')
    from math import pi
    s1_ = s1/pi
    fig.suptitle("$s\in[%1.f,%1.f\pi]$ , $u' = %.1f$ , $v' = %.1f$"%(s0,s1_,u0[1],u0[3]))
    if show == True:
        plt.show()
    return X,plt

def display_torus(u0,s0,s1,ds,a=1,c=2,solver=None,show=False):
    C = torus(a,c) # Find the Christoffel tensor for the torus
    X = solve(C,u0,s0,s1,ds,solver)

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    N = X[:,0].shape[0]
```

```
    u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
    x = (c + a*np.cos(v))*np.cos(u)
    y = (c + a*np.cos(v))*np.sin(u)
    z = np.sin(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(elev=-60, azim=100)
    # use transparent colormap -> negative
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = 2*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

    # plot the parametrized data on to the torus
    u,v = X[:,0], X[:,2]
    x = (c + a*np.cos(v))*np.cos(u)
    y = (c + a*np.cos(v))*np.sin(u)
    z = np.sin(v)

    ax.plot(x,y,z,'--r')
    s1_ = s1/pi
    fig.suptitle("$s\in[%1.f,%1.f\pi]$ , $u = %.1f$ , $u' = %.1f$, $v = %.1f$ , $v' = %.1f$"%(s0,s1_,u0
        [0],u0[1],u0[2],u0[3]))
    if show == True:
       plt.show()
    return X,plt

def display_toroid(u0,s0,s1,ds,u_val=1,v_val=None,a=1,solver=None,show=False):
    C = toroid(u_val,v_val,a) # Find the Christoffel tensor for toroid
    X = solve(C,u0,s0,s1,ds,solver)

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    from math import pi
    if v_val is None:
       u = u_val # toroids
       v,w = plt.meshgrid(np.linspace(-pi,pi,250),np.linspace(0,2*pi,250))
    else:
       v = v_val # spherical bowls
       u,w = plt.meshgrid(np.linspace(0,2,250),np.linspace(0,2*pi,250))

    x = (a*np.sinh(u)*np.cos(w))/(np.cosh(u) - np.cos(v))
    y = (a*np.sinh(u)*np.sin(w))/(np.cosh(u) - np.cos(v))
    z = (a*np.sin(v))/(np.cosh(u) - np.cos(v))

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(elev=90., azim=0)
    # use transparent colormap
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = -.5*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

    # plot the parametrized data on to the toroid
    if v_val is None:
       w,v = X[:,0], X[:,2]
    else:
       w,u = X[:,0], X[:,2]
    x = (a*np.sinh(u)*np.cos(w))/(np.cosh(u) - np.cos(v))
    y = (a*np.sinh(u)*np.sin(w))/(np.cosh(u) - np.cos(v))
```

```
    z = (a*np.sin(v))/(np.cosh(u) - np.cos(v))

    s1_ = s1/pi
    ax.plot(x,y,z,'--r')
    fig.suptitle('$s\in[%.1f\pi \, , \,%2.1f\pi]$'%(s0,s1_))
    if show == True:
        plt.show()
    return X,plt

def display_egg_carton(u0,s0,s1,ds,solver=None,show=False):
    C = egg_carton() # Find the Christoffel tensor for egg carton surface
    X = solve(C,u0,s0,s1,ds,solver)

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    from math import pi
    N = X[:,0].shape[0]
    u,v = plt.meshgrid(np.linspace(s0,s1,N),np.linspace(s0,s1,N))
    x = u
    y = v
    z = np.sin(u)*np.cos(v)

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(elev=90., azim=0)
    # use transparent colormap
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = -.5*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

    # plot the parametrized data on to the egg carton
    u,v = X[:,0], X[:,2]
    x = u
    y = v
    z = np.sin(u)*np.cos(v)

    s0_ = s0/pi
    s1_ = s1/pi
    ax.plot(x,y,z,'--r')
    fig.suptitle('$s\in[%.1f\pi \, , \,%2.1f\pi]$'%(s0_,s1_))
    if show == True:
        plt.show()
    return X,plt

def display_3D_Kerr(u0,s0,s1,ds,solver=None,show=True,args=None,multiple=True):
    if args == None:
        C = flat_kerr() # use default values
    else:
        a = args[0]
        G = args[1]
        M = args[2]
        C = flat_kerr(a,G,M) # Find the Christoffel tensor for 3D Kerr metric on 4D manifold

    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    if multiple is not True:
        X = solve(C,u0,s0,s1,ds,solver)
        r = X[:,0]
        theta = X[:,2]
        # for time independent kerr metric use :
        #phi = X[:,4]
        #x = r*np.sin(theta)*np.cos(phi)
```

```
        #y = r*np.sin(theta)*np.sin(phi)
        #z = r*np.cos(theta)
        #ax.plot(x,y,z,'b')
        t = X[:,4]
        x = r*np.sin(theta)
        y = r*np.cos(theta)
        z = t
        ax.plot(x,y,z,'b')

    if multiple is True:
        plt.hold('on')
        N = 50
        t = np.linspace(0.01,np.pi-.01,N)
        for i in range(N):
            u0[0] = np.sin(t[i])
            u0[2] = np.cos(t[i])
            if u0[0] < 0:
                u0[0] = -.71+u0[0]
            else:
                u0[0] = .71+u0[0]
            print 'i=%d'%i, u0
            X = solve(C,u0,s0,s1,ds,solver)
            r = X[:,0]
            theta = X[:,2]
            #phi = X[:,4]
            #x = r*np.sin(theta)*np.cos(phi)
            #y = r*np.sin(theta)*np.sin(phi)
            #z = r*np.cos(theta)
            #ax.plot(x,y,z,'b')
            t = X[:,4]
            x = r*np.sin(theta)
            y = r*np.cos(theta)
            z = t
            ax.plot(x,y,z,'b')

    ax.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    plt.show()

    return X,plt

def display_multiple_geodesics(u0,s0,s1,ds,surface,with_object=True):
    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    def solve_multiple(C,u0,s0,s1,ds,s):
        from sympy.abc import u,v
        print 'Running solver...'
        return sc.odeint(f,u0,s,args=(C,u,v))

    def display_multiple_catenoid(u0,s0,s1,ds,C,s):
        X = solve_multiple(C,u0,s0,s1,ds,s)
        plt.hold('on')
        # plot the parametrized data on to the catenoid
        u,v = X[:,0], X[:,2]
        x = np.cos(u) - v*np.sin(u)
        y = np.sin(u) + v*np.cos(u)
        z = v
        ax.plot(x,y,z,'--r')
        return plt

    def display_multiple_egg_carton(u0,s0,s1,ds,C,s):
        X = solve_multiple(C,u0,s0,s1,ds,s)
        plt.hold('on')
        # plot the parametrized data on to the egg carton
        u,v = X[:,0], X[:,2]
        x = u
        y = v
```

```
    z = np.sin(u)*np.cos(v)
    ax.plot(x,y,z,'--r')
    return plt

def display_multiple_sphere(u0,s0,s1,ds,C,s):
    X = solve_multiple(C,u0,s0,s1,ds,s)
    plt.hold('on')
    # plot the parametrized data on to the sphere
    u,v = X[:,0], X[:,2]
    x = np.cos(u)*np.cos(v)
    y = np.sin(u)*np.cos(v)
    z = np.sin(v)
    ax.plot(x,y,z,'--r')
    return plt

def display_multiple_torus(u0,s0,s1,ds,C,s):
    X = solve_multiple(C,u0,s0,s1,ds,s)
    plt.hold('on')
    # plot the parametrized data on to the sphere
    u,v = X[:,0], X[:,2]
    x = (2 + 1*np.cos(v))*np.cos(u)
    y = (2 + 1*np.cos(v))*np.sin(u)
    z = np.sin(v)
    ax.plot(x,y,z,'--r')
    return plt

u0_range = np.arange(s0,s1+ds,ds)
N = u0_range.shape[0]

fig = plt.figure()
if surface == 'catenoid':
    if with_object:
        u,v = plt.meshgrid(np.linspace(-np.pi,np.pi,150),np.linspace(-np.pi,np.pi,150))
        x = np.cos(u) - v*np.sin(u)
        y = np.sin(u) + v*np.cos(u)
        z = v
    C = catenoid()
elif surface == 'egg_carton':
    if with_object:
        u,v = plt.meshgrid(np.linspace(-4,4,250),np.linspace(-4,4,250))
        x = u
        y = v
        z = np.sin(u)*np.cos(v)
    C = egg_carton()
elif surface == 'sphere':
    if with_object:
        u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
        x = np.cos(u)*np.cos(v)
        y = np.sin(u)*np.cos(v)
        z = np.sin(v)
    C = sphere()
elif surface == 'torus':
    if with_object:
        u,v = plt.meshgrid(np.linspace(0,2*np.pi,150),np.linspace(0,2*np.pi,150))
        x = (2 + 1*np.cos(v))*np.cos(u)
        y = (2 + 1*np.cos(v))*np.sin(u)
        z = np.sin(v)
    C = torus()
ax = fig.add_subplot(111, projection='3d')
ax.view_init(azim=65, elev=67)
if with_object:
    theCM = 'Pastel1'
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
plt.hold('on')

if surface == 'catenoid':
    for u_val in u0_range:
        u0[3] = u_val
```

```
            plt = display_multiple_catenoid(u0,s0,s1,ds,C,u0_range)
    elif surface == 'egg_carton':
        for u_val in u0_range:
            u0[0] = u_val
            plt = display_multiple_egg_carton(u0,s0,s1,ds,C,u0_range)
    elif surface == 'sphere':
        for u_val in u0_range:
            u0[0] = u_val
            plt = display_multiple_sphere(u0,s0,s1,ds,C,u0_range)
    elif surface == 'torus':
        for u_val in u0_range:
            u0[0] = u_val # alternate v0 values
            plt = display_multiple_torus(u0,s0,s1,ds,C,u0_range)


    from math import pi
    s0_ = s0#/pi
    s1_ = s1/pi
    fig.suptitle("$s\in[%1.f,%1.f\pi]$ , $u' = %.1f$ , $v = %.1f$ , $v' = %.1f$"%(s0_,s1_,u0[1],u0[2],u0
        [3]))
    plt.show()

if __name__ == '__main__':
    import sys
    from math import pi, sqrt
    if len(sys.argv) > 1:
        u0 = eval(sys.argv[1]) # evaluate the input for a list [u(0), u'(0), v(0), v'(0)]
        if type(u0) is not list:
            raise TypeError("The first argument must be a list : [u(0),v(0),u'(0),v'(0)]")
        s0 = float(eval(sys.argv[2])) # evaluate math expressions such as pi, '*', sqrt
        s1 = float(eval(sys.argv[3]))
        ds = float(eval(sys.argv[4]))
        display = sys.argv[5]
        if display == 'catenoid':
            display_catenoid(u0,s0,s1,ds)
        if display == 'sphere':
            display_sphere(u0,s0,s1,ds)
        if display == 'torus':
            display_torus(u0,s0,s1,ds)
        if display == 'egg_carton':
            display_egg_carton(u0,s0,s1,ds)
    else:
        """
        u0 = [0,-.5,.5,0] # u(0), u'(0), v(0), v'(0)
        s0 = -pi/2
        s1 = 3*pi
        ds = 0.1
        display_catenoid(u0,s0,s1,ds,show=True)
        #display_multiple_geodesics(u0,s0,s1,ds,'catenoid',with_object=False)

        u0 = [0.75,0.1,.75,0.1]
        s0 = 0
        s1 = 18*pi
        ds = 2
        #display_sphere(u0,s0,s1,ds,show=True)
        display_multiple_geodesics(u0,s0,s1,ds,'sphere',with_object=False)

        u0 = [0,.2,0,.2] # u(s0), u'(s0), v(s0), v'(s0)
        s0 = 0
        s1 = 25*pi
        ds = .1
        a = 1
        c = -2
        display_torus(u0,s0,s1,ds,a=a,c=c,show=True)
        #display_multiple_geodesics(u0,s0,s1,ds,'torus',with_object=False)

        u0 = [0,.5,0.5,sqrt(3)/2] # u(0), u'(0), v(0), v'(0)
        s0 = -pi
```

```
        s1 = pi
        ds = 0.05
        display_egg_carton(u0,s0,s1,ds,show=True)
        #display_multiple_geodesics(u0,s0,s1,ds,'egg_carton')

        u0 = [1.5,.1,-1,0] # u(0), u'(0), v(0), v'(0)
        s0 = 0
        s1 = 80
        ds = 0.1
        display_mobius_strip(u0,s0,s1,ds,show=True,solver=None)


        u0 = [0,0,0,.1] # u(s0), u'(s0), v(s0), v'(s0)
        s0 = 0
        s1 = 10*pi
        ds = 0.5
        display_toroid(u0,s0,s1,ds,u_val=1,show=True)

        # check if two points are connected on great circle
        p1 = (1,1) # taken from a sphere simulation for u' = .2, v' = 0
        p2 = (0.16242917238268037, 0.80611950949349132) # entry 200 in X
        s0 = 0
        s1 = 18*pi
        ds = 0.05
        C = sphere()
        two_points(p1,p2,s0,s1,ds,C,tol=1e-6,surface='sphere')
        """
        u0 = [.7,.1,.1,.1,0,.1] # u(s0), u'(s0), v(s0), v'(s0), w(s0), w'(s0)
        s0 = 0
        s1 = 18
        ds = 0.01
        # A singularity near origo : a = 0, G = 1, M = 0.5
        a = 0
        G = 1
        M = 0.35
        display_3D_Kerr(u0,s0,s1,ds,show=True,solver=None,args = (a,G,M))
```

C.6 Hyperstreamlines

hyperstreamlines.py

```
from __future__ import division
import scipy.integrate as sc
import numpy as np
import numpy.linalg as nplin

def find_eigen(T):
    """
    Returns the eigenvalues and eigenvectors for a second order tensor T.
    """
    dim = T.shape[0]
    eig_data = nplin.eig(T)
    eig_values = eig_data[0]
    v1 = eig_data[1][:,0]
    v2 = eig_data[1][:,1]
    if dim == 3:
        v3 = eig_data[1][:,2]
        return eig_values,v1,v2,v3
    else:
        return eig_values,v1,v2

from scipy.interpolate import griddata
def integrate(grid_points,U,p0,s,direction='major',solver=None):
    dim = 2
    if len(U) == 12:
        dim = 3
        U_x, U_y, U_z, U_x_, U_y_, U_z_, U_x__, U_y__, U_z__,l_minor, l_major, l_medium = U
```

```
        if direction == 'major':
            U__x = U_x.flatten(); U__y = U_y.flatten(); U__z = U_z.flatten();
        else:
            U__x = U_x_.flatten(); U__y = U_y_.flatten(); U__z = U_z_.flatten();
        points = zip(grid_points[0].flatten(),grid_points[1].flatten(),grid_points[2].flatten())
        def f_3D(x,t):
            return [griddata(points,U__x,x)[0], griddata(points,U__y,x)[0], griddata(points,U__z,x)[0]]
        f = f_3D
    else:
        U_x, U_y, U_x_, U_y_, l_minor, l_major = U
        if direction == 'major':
            U__x = U_x.flatten(); U__y = U_y.flatten();
        else:
            U__x = U_x_.flatten(); U__y = U_y_.flatten();
        points = zip(grid_points[0].flatten(),grid_points[1].flatten())
        def f_2D(x,t):
            return [griddata(points,U__x,x)[0], griddata(points,U__y,x)[0]]
        f = f_2D

    if solver == None: # use lsoda from scipy.integrate.odeint
        print 'Running solver ...'
        p = sc.odeint(f,p0,s)
    else: # use any other solver from scipy.integrate.ode
        # vode,zvode,lsoda,dopri5,dop853
        r = sc.ode(lambda t,x: f(x,t)).set_integrator(solver)
        r.set_initial_value(p0)
        y = []
        t_end = s[-1]
        dt = s[1]-s[0]
        print 'Running solver ...'
        while r.successful() and r.t <= t_end:
            r.integrate(r.t + dt)
            y.append(r.y)
            p = np.array(y)
    if direction == 'major':
        if dim == 2:
            p_ = _expand_hyperstreamline(p,U_x_,U_y_,l_minor,points)
        else:
            p_ = _expand_hyperstreamline3D(p,U_x_,U_y_,U_z_,U_x__,U_y__,U_z__,l_minor,l_medium,points)
    else:
        if dim == 2:
            p_ = _expand_hyperstreamline(p,U_x,U_y,l_major,points)
        else:
            p_ = _expand_hyperstreamline3D(p,U_x,U_y,U_z,U_x__,U_y__,U_z__,l_major,l_medium,points)
    return p,p_

def _expand_hyperstreamline(p,Ux,Uy,l,points):
    p_ = np.zeros_like(p)
    def f(x):
        return [griddata(points,Ux,x)[0], griddata(points,Uy,x)[0]]
    i = 0
    for p_val in p:
        print p_val
        p_[i] = f(p_val)*l[p_val]
        i = i + 1
    return p_


def _expand_hyperstreamline3D(p,Ux,Uy,Uz,Ux_,Uy_,Uz_,l,l_,points):
    p_ = np.zeros_like(p)
    def f1(x):
        return [griddata(points,Ux,x)[0], griddata(points,Uy,x)[0],griddata(points,Uz,x)[0]]
    def f2(x):
        return [griddata(points,Ux_,x)[0], griddata(points,Uy_,x)[0],griddata(points,Uz_,x)[0]]
    return p_

def extract_eigen(eigen_field):
```

```
    """
    Performs a sorting of minor, major, and (if 3D) medium eigenvectors and eigenvalues.
    The 'eigen_field' is assumed to be of the form

                 [[ l1, l2],
                  [v1x, v1y],
                  [v2x, v2y]]
    and for 3D
                 [[ l1, l2, l3],
                  [v1x, v1y, v1z],
                  [v2x, v2y, v2z],
                  [v3x, v3y, v3z]]
    Finally, the corresponding eigenvalues are returned as well.
    """
    return _sort_eig(eigen_field)

def _sort_eig(U):
    dim = U.shape[1]
    Nx = U.shape[2]
    Ny = U.shape[3]
    if dim == 3:
        Nz = U.shape[4]
        return _sort_eig_3D(U,Nx,Ny,Nz)

    U_x = np.zeros([Nx,Ny]); U_y = np.zeros_like(U_x); # major eigenvectors
    U_x_ = np.zeros_like(U_x); U_y_ = np.zeros_like(U_x); # minor eigenvectors
    l_major = np.zeros_like(U_x); l_minor = np.zeros_like(U_x)
    print 'Sorting eigenvalues and eigenvectors ...'
    for i in range(Nx):
        for j in range(Ny):
            if U[0,0,i,j] <= U[0,1,i,j]: # if lambda_1 < lambda_2
                l_minor[i,j] = U[0,0,i,j]
                l_major[i,j] = U[0,1,i,j]
                U_x[i,j] = U[2,0,i,j]
                U_y[i,j] = U[2,1,i,j]
                U_x_[i,j] = U[1,0,i,j]
                U_y_[i,j] = U[1,1,i,j]
            else:
                l_major[i,j] = U[0,1,i,j]
                l_minor[i,j] = U[0,0,i,j]
                U_x[i,j] = U[1,0,i,j]
                U_y[i,j] = U[1,1,i,j]
                U_x_[i,j] = U[2,0,i,j]
                U_y_[i,j] = U[2,1,i,j]
    return U_x_, U_y_, U_x, U_y, l_minor, l_major

def _sort_eig_3D(U,Nx,Ny,Nz):
    U_x = np.zeros([Nx,Ny,Nz]); U_y = np.zeros_like(U_x); U_z = np.zeros_like(U_x); # major
    U_x_ = np.zeros_like(U_x); U_y_ = np.zeros_like(U_x); U_z_ = np.zeros_like(U_x); # minor
    U_x__ = np.zeros_like(U_x); U_y__ = np.zeros_like(U_x); U_z__ = np.zeros_like(U_x); # medium
    l_major = np.zeros_like(U_x); l_minor = np.zeros_like(U_x); l_medium = np.zeros_like(U_x);
    print 'Sorting eigenvalues and eigenvectors ...'
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                if (U[0,0,i,j,k] >= U[0,1,i,j,k]) and (U[0,0,i,j,k] >= U[0,2,i,j,k]):
                    l_major[i,j,k] = U[0,0,i,j,k]
                    if U[0,1,i,j,k] >= U[0,2,i,j,k]:
                        l_minor[i,j,k] = U[0,2,i,j,k]
                        l_medium[i,j,k] = U[0,1,i,j,k]
                        U_x_[i,j,k] = U[2,0,i,j,k]
                        U_y_[i,j,k] = U[2,1,i,j,k]
                        U_z_[i,j,k] = U[2,2,i,j,k]
                        U_x__[i,j,k] = U[3,0,i,j,k]
                        U_y__[i,j,k] = U[3,1,i,j,k]
                        U_z__[i,j,k] = U[3,2,i,j,k]
                    else:
                        l_minor[i,j,k] = U[0,1,i,j,k]
```

```
                        l_medium[i,j,k] = U[0,2,i,j,k]
                        U_x_[i,j,k] = U[3,0,i,j,k]
                        U_y_[i,j,k] = U[3,1,i,j,k]
                        U_z_[i,j,k] = U[3,2,i,j,k]
                        U_x__[i,j,k] = U[2,0,i,j,k]
                        U_y__[i,j,k] = U[2,1,i,j,k]
                        U_z__[i,j,k] = U[2,2,i,j,k]
                    U_x[i,j,k] = U[1,0,i,j,k]
                    U_y[i,j,k] = U[1,1,i,j,k]
                    U_z[i,j,k] = U[1,2,i,j,k]
                elif (U[0,1,i,j,k] >= U[0,0,i,j,k]) and (U[0,1,i,j,k] >= U[0,2,i,j,k]):
                    l_major[i,j,k] = U[0,1,i,j,k]
                    if U[0,0,i,j,k] >= U[0,2,i,j,k]:
                        l_minor[i,j,k] = U[0,2,i,j,k]
                        l_medium[i,j,k] = U[0,0,i,j,k]
                        U_x_[i,j,k] = U[3,0,i,j,k]
                        U_y_[i,j,k] = U[3,1,i,j,k]
                        U_z_[i,j,k] = U[3,2,i,j,k]
                        U_x__[i,j,k] = U[1,0,i,j,k]
                        U_y__[i,j,k] = U[1,1,i,j,k]
                        U_z__[i,j,k] = U[1,2,i,j,k]
                    else:
                        l_minor[i,j,k] = U[0,0,i,j,k]
                        l_medium[i,j,k] = U[0,2,i,j,k]
                        U_x_[i,j,k] = U[1,0,i,j,k]
                        U_y_[i,j,k] = U[1,1,i,j,k]
                        U_z_[i,j,k] = U[1,2,i,j,k]
                        U_x__[i,j,k] = U[3,0,i,j,k]
                        U_y__[i,j,k] = U[3,1,i,j,k]
                        U_z__[i,j,k] = U[3,2,i,j,k]
                    U_x[i,j,k] = U[2,0,i,j,k]
                    U_y[i,j,k] = U[2,1,i,j,k]
                    U_z[i,j,k] = U[2,2,i,j,k]
                else:
                    l_major[i,j,k] = U[0,2,i,j,k]
                    if U[0,0,i,j,k] >= U[0,1,i,j,k]:
                        l_minor[i,j,k] = U[0,1,i,j,k]
                        l_medium[i,j,k] = U[0,0,i,j,k]
                        U_x_[i,j,k] = U[2,0,i,j,k]
                        U_y_[i,j,k] = U[2,1,i,j,k]
                        U_z_[i,j,k] = U[2,2,i,j,k]
                        U_x__[i,j,k] = U[1,0,i,j,k]
                        U_y__[i,j,k] = U[1,1,i,j,k]
                        U_z__[i,j,k] = U[1,2,i,j,k]
                    else:
                        l_minor[i,j,k] = U[0,0,i,j,k]
                        l_medium[i,j,k] = U[0,1,i,j,k]
                        U_x_[i,j,k] = U[1,0,i,j,k]
                        U_y_[i,j,k] = U[1,1,i,j,k]
                        U_z_[i,j,k] = U[1,2,i,j,k]
                        U_x__[i,j,k] = U[2,0,i,j,k]
                        U_y__[i,j,k] = U[2,1,i,j,k]
                        U_z__[i,j,k] = U[2,2,i,j,k]
                    U_x[i,j,k] = U[3,0,i,j,k]
                    U_y[i,j,k] = U[3,1,i,j,k]
                    U_z[i,j,k] = U[3,2,i,j,k]
    return U_x, U_y, U_z, U_x_, U_y_, U_z_, U_x__, U_y__, U_z__,l_minor, l_major, l_medium

def _run_example_flat_sphere(xstart,xend,N,direction='major',solver=None):
    """
    A test example, using the metric of a flat sphere, to calculate hyperstreamlines
    for a 2D grid.
    """
    x0,y0 = xstart
    xN,yN = xend
    Nx,Ny = N
    x,y = np.mgrid[x0:xN:Nx*1j,y0:yN:Ny*1j]
    # Initialize the metric for the flat sphere
```

```
    g = np.array([[1,0],[0,1]],dtype=np.float32)
    T = np.zeros([2,2,Nx,Ny],dtype=np.float32) # The tensor field
    eig_field = np.zeros([3,2,Nx,Ny],dtype=np.float32) # The "eigen" field

    print "Determining eigenvectors for the flat metric of a sphere over the mesh..."
    for i in range(Nx):
        for j in range(Ny):
            g[1,1]= np.sin(y[i,j])**2
            T[:,:,i,j] = g[:,:]
            eig_field[:,:,i,j] = find_eigen(T[:,:,i,j])

    INITIAL_POINT = (1.,1.)
    t0 = 0
    t1 = 2*np.pi
    dt = 0.01
    t = np.arange(t0,t1+dt,dt)
    U = extract_eigen(eig_field)
    p,p_ = integrate([x,y],U,INITIAL_POINT,t,direction=direction,solver=solver)
    return p,p_

def _run_example_3D(xstart,xend,N,direction=’major’,solver=None):
    """
    A 3D test example
    """
    x0,y0,z0 = xstart
    xN,yN,yN = xend
    Nx,Ny,Nz = N
    x,y,z = np.mgrid[x0:xN:Nx*1j,y0:yN:Ny*1j,z0:zN:Nz*1j]
    # Initialize the metric for the flat sphere
    g = np.array([[1,0,0],[0,.5,0],[0,0,1]],dtype=np.float32)
    T = np.zeros([3,3,Nx,Ny,Nz],dtype=np.float32) # The tensor field
    eig_field = np.zeros([4,3,Nx,Ny,Nz],dtype=np.float32) # The "eigen" field

    print "Determining eigenvectors for the flat metric of a sphere over the mesh..."
    for i in range(Nx):
        for j in range(Ny):
            for k in range(Nz):
                g[2,2]= np.sin(y[i,j,k])**2 + np.cos(x[i,j,k])**2
                T[:,:,i,j,k] = g[:,:]
                eig_field[:,:,i,j,k] = find_eigen(T[:,:,i,j,k])

    INITIAL_POINT = (1.,1.,1.)
    t0 = 0
    t1 = 2*np.pi
    dt = 0.01
    t = np.arange(t0,t1+dt,dt)
    U = extract_eigen(eig_field)
    p,p_ = integrate([x,y,z],U,INITIAL_POINT,t,direction=direction,solver=solver)
    return p,p_

if __name__ == "__main__":
    import sys
    x0 = 0; y0 = 0; z0 = 0
    xN = np.pi/2; yN = np.pi; zN = 1
    Nx = 22; Ny = 22; Nz = 22
    N = (Nx,Ny,Nz)#N = (Nx,Ny)
    xstart = (x0,y0,z0); xend = (xN,yN,zN)
    #xstart = (x0,y0); xend = (xN,yN)
    solver = None # solvers: lsoda (default), vode,zvode,lsoda,dopri5,dop853

    if len(sys.argv) > 1:
        if sys.argv[1] == "major":
            p,p_= _run_example_flat_sphere(xstart,xend,N,’major’,solver=solver)
        else:
            p,p_= _run_example_flat_sphere(xstart,xend,N,’minor’,solver=solver)
    else:
        #p,p_= _run_example_flat_sphere(xstart,xend,N,’major’,solver=solver)
        p,p_= _run_example_3D(xstart,xend,N,’major’,solver=solver)
```

```python

import matplotlib.pylab as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
plt.plot(p[:,0],p[:,1],p[:,2])
plt.show()
```