

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 multiplication¹. The entire vector space is spanned by the orthogonal basis vectors.

Definition. *Let the orthogonal 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*

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

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 referring 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, \dots, v_N) . Dots imply the remaining components of the vector, instead of listing them all up. The vector space is spanned by the basis $\{\hat{e}_1, \dots, \hat{e}_N\}$.

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

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \dots + v_N^2}. \quad (2)$$

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 $\mathbf{x} = (x_1, \dots, x_N)$. 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.

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

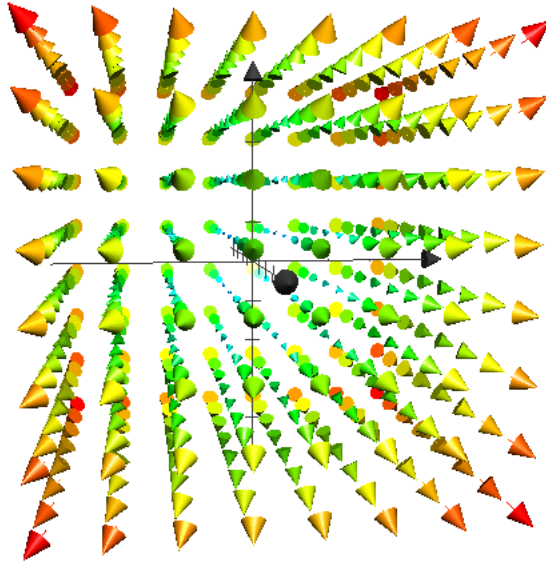


Figure 1: Vector field (x,y,z) displayed using a simple graphical tool in OS X.

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 instantaneous 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 moment¹. Mathematically, we can determine the path by solving the differential equations

$$\frac{dx_1}{v_1(\mathbf{x})} \Big|_{t=t_0} = \frac{dx_2}{v_2(\mathbf{x})} \Big|_{t=t_0} = \frac{dx_3}{v_3(\mathbf{x})} \Big|_{t=t_0} \quad (3)$$

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 :

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

1. Discretize the domain $\mathbf{x} \in [0, x_{\max}] \times [0, y_{\max}]$ where the vector field exists.
2. Use a start point (x_0, 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

- 1: *Load vector field data into arrays*
 - 2: *Create a grid over a domain for which the vector field exists upon*
 - 3: *Create arrays for storage of the streamline data*
 - 4: *Perform numerical integration by the forward Euler method*
 - 5: *Repeat process for backwards integration*
 - 6: *Stitch together all the entries for forward and backwards integration*
-

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

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

where $i, j \in \mathbb{N}$. 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 ($N \times N$) 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 *principal axes*.

We can always factorize a diagonalizable¹ square ($N \times N$) matrix with N eigenvectors

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

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

```

1 import numpy as np
2
3 def T1(x_):
4     x = x_[0]
5     y = x_[1]
6     return np.array([[.5*x**2 + 2*x*y + .5*y**2, -x**2 + y**2],
7                     [-x**2 + y**2, -.5*x**2 - 2*x*y - 5*y**2]])
8
9 def T2(x_):
10     x = x_[0]
11     y = x_[1]
12     return np.array([[x**2 - y**2, 2*x*y],
13                     [2*x*y, x**2 - y**2]])
14
15 def T3(x_):
16     x = x_[0]
17     y = x_[1]
18     return np.array([[x**2 - 3*y**2, -5*x*y + 4*y**2],
19                     [-5*x*y + 4*y**2, x**2 - 3*y**2]])
20
21 def T4(x_):
22     x = x_[0]
23     y = x_[1]
24     return np.array([[x**4 - .5*x**2*y**2, 2*x**4 - 5*x**3*y - 9*x*y**3],
25                     [2*x**4 - 5*x**3*y - 9*x*y**3, x**4 - .5*x**2*y**2]])
26
27 def T5(x_):
28     x = x_[0]
29     y = x_[1]
30     return np.array([[-x**2 + y**2, -x**2 - 2*x*y + y**2],
31                     [-x**2 - 2*x*y + y**2, -x**2 + y**2]])
32
33 def degenerate(T, tol=1e-12):
34     """
35     Function assumes that the tensor is given with the following form :
36
37     If the tensor values are distributed over a 100x100 grid, then for 2D,
38     (100,100,2,2) is the shape of the tensor T. Meaning, each grid point
39     contains a 2 dimensional second rank tensor. We assume that all such
40     tensors are symmetric. For a 3D tensor defined over a 16x50x16 grid,

```

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

```

38 T is of the shape (16,50,16,3,3). Here too we assume that all grid
39 tensors are symmetric.
40 """
41 if T.shape[-2:] == (2,2):
42     if len(T.shape[:-2]) == 2:
43         deg_points = _find_all_degen_points(T,dim=2,tol=tol)
44         return deg_points
45 elif T.shape[-2:] == (3,3):
46     if len(T.shape[:-2]) == 3:
47         deg_points = _find_all_degen_points(T,dim=3,tol=tol)
48         return deg_points
49 msg = "Tensor has wrong shape.\n"
50 msg += "Tensor shape must be either in 3D\n"
51 msg += "(data x, data y, data z, gridx, gridy, gridz)\n"
52 msg += "or in 2D\n(data x, data y, gridx, gridy)"
53 raise IndexError(msg)
54
55 def _find_all_degen_points(T,dim,tol):
56     if tol>1e-6:
57         print "Warning : Tolerance value provided is too large."
58     xdata = T.shape[0]
59     ydata = T.shape[1]
60     if dim == 3:
61         zdata = T.shape[2]
62     fdp = 0 # counter for the amount of times we find a degenerate point
63
64     class SparseMatrix:
65         def __init__(self):
66             self.entries = {}
67
68         def __call__(self, tuple, value=0):
69             self.entries[tuple] = value
70
71         def value(self, tuple):
72             try:
73                 value = self.entries[tuple]
74             except KeyError:
75                 value = 0
76             return value
77
78     deg_points = SparseMatrix()
79     if dim == 2:
80         """
81         Solve following system of equations for a 2D tensor :
82          $T[i,j,0,0] - T[i,j,1,1] == 0$ 
83          $T[i,j,0,1] == 0$ 
84         """
85         found = np.zeros(2)
86         for i in range(xdata):
87             for j in range(ydata):
88                 found[0] = np.fabs(T[i,j,0,0] - T[i,j,1,1]) <= tol
89                 found[1] = np.fabs(T[i,j,0,1]) <= tol
90
91                 if np.all(found):
92                     deg_points((i,j,k),1)
93                     fdp = fdp + 1
94     elif dim == 3:
95         """
96         Solve following system of equations for a 3D tensor :
97          $T[i,j,k,0,0] - T[i,j,1,1] == 0$ 
98          $T[i,j,k,1,1] - T[i,j,2,2] == 0$ 
99          $T[i,j,k,0,1] == 0$ 
100         $T[i,j,k,0,2] == 0$ 
101         $T[i,j,k,1,2] == 0$ 
102        """
103        found = np.zeros(5)
104        for i in range(xdata):
105            for j in range(ydata):

```

```

106         for k in range(zdata):
107             found[0] = np.fabs(T[i,j,k,0,0] - T[i,j,k,1,1]) <= tol
108             found[1] = np.fabs(T[i,j,k,1,1] - T[i,j,k,2,2]) <= tol
109             found[2] = np.fabs(T[i,j,k,0,1]) <= tol
110             found[3] = np.fabs(T[i,j,k,0,2]) <= tol
111             found[4] = np.fabs(T[i,j,k,1,2]) <= tol
112
113             if np.all(found):
114                 deg_points((i,j,k),1)
115                 fdp = fdp + 1
116
117     print "Found %d degenerate points in the tensor data." %(fdp)
118     return deg_points, fdp

```

C.2 Invariance

invariance.py

```

1 import sympy
2 from sympy import diff
3 import numpy
4
5 def msg(case,T,x):
6     dim = len(x)
7     T = numpy.array(T)
8     if case:
9         print "\nThe tensor"
10        print T
11        if dim==2:
12            print "has a degenerate point at (%d,%d)" %(x[0],x[1])
13        if dim==3:
14            print "has a degenerate point at (%d,%d,%d)" %(x[0],x[1],x[2])
15    else:
16        print "\nThe Tensor"
17        print T
18        if dim==2:
19            print "does not have a degenerate point at (%d,%d)" %(x[0],x[1])
20        if dim==3:
21            print "does not have a degenerate point at (%d,%d,%d)" %(x[0],x[1],x[2])
22
23 def invariant2D(T,coords,info=False):
24     """
25     Function assumes that the tensor is given in analytical form as a
26     sympy matrix. Further it only considers 2D second rank symmetric
27     tensors, i.e of the form
28
29         T = [T11 T12; T12 T22]
30
31     The invariant of the tensor is determined :
32         delta = ad - bc
33     where a = d/dx(T11 - T22), d = d/dy(T12), b = d/dy(T11 - T22),
34     and c = d/dx(T12). If the invariant is found to be zero, the
35     tensor has two equal eigenvalues, i.e the tensor is degenerate.
36
37     This technique is based on PhD thesis of Delmarcelle - see Del94
38     """
39     x0 , y0 = coords[0], coords[1]
40
41     T11 = 0.5*T[0,0]
42     T12 = 0.5*T[0,1]
43     T22 = 0.5*T[1,1]
44
45     x, y = sympy.symbols('x y')
46
47     delta = \
48     diff(T11-T22,x).evalf(subs={x:x0, y:y0})*diff(T12,y).evalf(subs={x:x0, y:y0})
49     -\

```

```

50     diff(T11-T22,y).evalf(subs={x:x0, y:y0})*diff(T12,x).evalf(subs={x:x0, y:y0})
51
52     eps = 1e-12
53     if abs(delta) <= eps :
54         if info:
55             msg(1,T,coords)
56         return 1
57     else:
58         if info:
59             msg(0,T,coords)
60         return 0
61
62 def invariant3D_discriminant(T,coords,info=True):
63     """
64     Function assumes that the tensor is given in analytical form as a
65     sympy matrix. Further it only considers 3D second rank symmetric
66     tensors, i.e of the formd
67
68         T = [T00 T01 T02;
69              T01 T11 T12;
70              T02 T12 T22]
71
72     The discriminant of the tensor is evaluated. If it equals zero, this implies
73     that the tensor has at least two equal eigenvalues, i.e the tensor is
74     degenerate.
75
76     This technique is based on the article by Zheng et. al. - see ZTP06
77     """
78     x0, y0, z0 = coords[0], coords[1], coords[2]
79     x,y,z = sympy.symbols('x y z')
80
81     P = T[0,0] + T[1,1] + T[2,2]
82     Q = sympy.det(T[:2,:2]) + sympy.det(T[1:,1:]) + T[2,2]*T[0,0] - T[0,2]*T[0,2]
83     R = sympy.det(T)
84
85     D = Q**2*P**2 - 4*R*P**3 - 4*Q**3 - 4*Q**3 + 18*P*Q*R - 27*R**2
86
87     D_value = D.evalf(subs={x:x0,y:y0,z:z0})
88     eps = 1e-12
89     if abs(D_value) <= eps :
90         if info:
91             msg(1,T,coords)
92         return 1
93     else:
94         if info:
95             msg(0,T,coords)
96         return 0
97
98 def invariant3D_constraint_functions(T,coords,info=True):
99     """
100     This function is another representation of the discriminant :
101     see invariant3D_discriminant().
102
103     Function assumes that the tensor is given in analytical form as a
104     sympy matrix. Further it only considers 3D second rank symmetric
105     tensors, i.e of the formd
106
107         T = [T00 T01 T02;
108              T01 T11 T12;
109              T02 T12 T22]
110
111     The constaint function of the tensor is evaluated. If it equals zero,
112     this implies that the tensor has at least two equal eigenvalues, i.e
113     the tensor is degenerate.
114
115     This technique is based on the article by Zheng et. al. - see ZTP06
116     """
117     x0, y0, z0 = coords[0], coords[1], coords[2]

```

```

118 x,y,z = sympy.symbols('x y z')
119
120 fx = T[0,0]*(T[1,1]**2 - T[2,2]**2) + T[0,0]*(T[0,1]**2 - T[0,2]**2)\
121      + T[1,1]*(T[2,2]**2 - T[0,0]**2) + T[1,1]*(T[1,2]**2 - T[0,1]**2)\
122      + T[2,2]*(T[0,0]**2 - T[1,1]**2) + T[2,2]*(T[0,2]**2 - T[1,2]**2)
123
124 fy1 = T[1,2]*(2*(T[1,2]**2 - T[0,0]**2) - (T[0,2]**2 + T[0,1]**2)\
125      + 2*(T[1,1]*T[0,0] + T[2,2]*T[0,0] - T[1,1]*T[2,2]))\
126      + T[0,1]*T[0,2]*(2*T[0,0] - T[2,2] - T[1,1])
127
128 fy2 = T[0,2]*(2*(T[0,2]**2 - T[1,1]**2) - (T[0,1]**2 + T[1,2]**2)\
129      + 2*(T[2,2]*T[1,1] + T[0,0]*T[1,1] - T[2,2]*T[0,0]))\
130      + T[1,2]*T[0,1]*(2*T[1,1] - T[0,0] - T[2,2])
131
132 fy3 = T[0,1]*(2*(T[0,1]**2 - T[2,2]**2) - (T[1,2]**2 + T[0,2]**2)\
133      + 2*(T[0,0]*T[2,2] + T[1,1]*T[2,2] - T[0,0]*T[1,1]))\
134      + T[0,2]*T[1,2]*(2*T[2,2] - T[1,1] - T[0,0])
135
136 fz1 = T[1,2]*(T[0,2]**2 - T[0,1]**2) + T[0,1]*T[0,2]*(T[1,1] - T[2,2])
137
138 fz2 = T[0,2]*(T[0,1]**2 - T[1,2]**2) + T[1,2]*T[0,1]*(T[2,2] - T[0,0])
139
140 fz3 = T[0,1]*(T[1,2]**2 - T[0,2]**2) + T[0,2]*T[1,2]*(T[0,0] - T[1,1])
141
142 D = fx**2 + fy1**2 + fy2**2 + fy3**2 + 15*fz1**2 + 15*fz2**2 + 15*fz3**2
143
144 D_value = D.evalf(subs={x:x0,y:y0,z:z0})
145 eps = 1e-12
146 if abs(D_value) <= eps :
147     if info:
148         msg(1,T,coords)
149     return 1
150 else:
151     if info:
152         msg(0,T,coords)
153     return 0
154
155 if __name__ == "__main__":
156     x,y,z = sympy.symbols('x y z')
157
158     T1 = sympy.Matrix([[ 0.5*x**2, -x**2+y**2 ],
159                        [-x**2+y**2, -0.5*x**2 - 2*x*y - 0.5*y**2]])
160
161     x0 = 0
162     y0 = 0
163     coords = (x0,y0)
164     invariant2D(T1,coords,info=True)
165
166     x0 = 1
167     y0 = 1
168     z0 = 1
169     coords = (x0,y0,z0)
170     T2 = sympy.Matrix([[x**2,x*y,y**2],
171                        [x*y,y**2,y*z],
172                        [x**2,y*z,z**2]])
173     discrim = invariant3D_discriminant(T2,coords,info=True)
174     constrain = invariant3D_constraint_functions(T2,coords,info=True)
175
176     if discrim == constrain:
177         print "Both functions give same result!"

```

C.3 Find the Metric g_{ij}

find_metric.py

```

1 import sympy as sym
2
3 def curve_to_metric(ds2,dim,diff_=None):

```



```

4     if dim < 2:
5         raise ValueError("The metric is implemented for at least dim = 2.")
6     ds2 = str(ds2)
7     ds2 = ds2.replace(' ','')
8     differentials = []
9     if diff_ is None:
10        if dim >= 2:
11            differentials = ['du','dv']
12        if dim >= 3:
13            differentials.append('dw')
14        if dim == 4:
15            differentials.append('dt')
16    else:
17        M = sym.Matrix(diff_)
18        for i in range(0,M.shape[0]):
19            n = str(M[i,i]).find('*')
20            diff = str(M[i,i])[0:n]
21            differentials.append(diff)
22
23    for diff in differentials:
24        ds2 = ds2.replace(diff+'**2',diff+'*'+diff)
25
26    def split_elements(expr,tmp_list,side='right'):
27        new_list = []
28        if type(tmp_list) is list:
29            for term in tmp_list:
30                split_terms = term.split(expr)
31                if type(split_terms) is list:
32                    for sterms in split_terms:
33                        new_list.append(sterms)
34                else:
35                    new_list.append(split_terms)
36        else:
37            split_terms = tmp_list.split(expr) # split at found expression
38            if type(split_terms) is list:
39                for sterms in split_terms:
40                    new_list.append(sterms)
41            else:
42                new_list.append(split_terms)
43
44        lost = expr[:-1]
45        sign = '-' # special case to be handled
46        if side=='left':
47            lost = expr[1:]
48        for i in range(len(new_list)):
49            term = new_list[i]
50            if side=='left':
51                first = 0
52                if term[first] == '*':
53                    new_list[i] = lost+new_list[i]
54            else:
55                last = len(term) - 1
56                if term[last] == '*':
57                    new_list[i] = new_list[i]+lost
58                if expr[-1] == sign: # if expression contains '-' at end
59                    # for next in list add '-'
60                    new_list[i+1] = '-'+new_list[i+1]
61        return new_list
62
63    n = ds2
64    L='left'
65    R='right'
66    p = '+'
67    m = '-'
68    for diff in differentials: # for each differential : dx_i = du,dv,dw,dt
69        expr = diff+p # 'dx_i+'
70        n = split_elements(expr,n,R)
71        expr = diff+m # 'dx_i-'

```

```

72     n = split_elements(expr,n,R)
73     expr = p+diff # '+dx_i'
74     n = split_elements(expr,n,L)
75     expr = m+diff # '-dx_i'
76     n = split_elements(expr,n,L)
77
78     # define the matrix structure of the metric components for mapping the
79     if dim == 2: # curve elements to their corresponding location
80         if diff_ is None:
81             diff = [['du*du','du*dv'],
82                     ['dv*du','dv*dv']]
83         else:
84             diff = diff_
85     if dim == 3:
86         if diff_ is None:
87             diff = [['du*du','du*dv','du*dw'],
88                     ['dv*du','dv*dv','dv*dw'],
89                     ['dw*du','dw*dv','dw*dw']]
90         else:
91             diff = diff_
92     if dim == 4:
93         if diff_ is None:
94             diff = [['du*du','du*dv','du*dw','dt*du'],
95                     ['dv*du','dv*dv','dv*dw','dv*dt'],
96                     ['dw*du','dv*dw','dw*dw','dw*dt'],
97                     ['dt*du','dt*dv','dt*dw','dt*dt']]
98         else:
99             diff = diff_
100     # add the elements in g without the above differentials
101     elements = n
102     g = [['0' for _ in range(dim)] for _ in range(dim)]
103     for element in elements:
104         for i in range(dim):
105             for j in range(dim):
106                 if (element.find(diff[i][j]) != -1):
107                     if (element.find('*' + diff[i][j]) != -1):
108                         g[i][j] = element.replace('*'+ diff[i][j],'')
109                     else:
110                         g[i][j] = element.replace(diff[i][j],'1')
111                 g[j][i] = g[i][j]
112     from sympy import Matrix, sin, cos, exp, log, cosh, sinh, sqrt, tan, tanh
113     from sympy.abc import u,v,w,t
114     return Matrix(g)
115
116
117 def metric(coord1,coord2,form="simplified",write_to_file=False):
118     """
119     Calculates the metric for the coordinate transformation
120     between cartesian coordinates to another orthogonal
121     coordinate system.
122     """
123     from sympy import diff
124     x,y = coord1[0], coord1[1]
125     u,v = coord2[0], coord2[1]
126     dim = len(coord1)
127     if len(coord2) != dim:
128         import sys
129         sys.exit("Coordinate systems must have same dimensions.")
130     if dim >= 3:
131         z = coord1[2]
132         w = coord2[2]
133     if dim == 4:
134         t1 = coord1[3]
135         t2 = coord2[3]
136     dxdu = diff(x,u)
137     dxdv = diff(x,v)
138     dydu = diff(y,u)
139     dydv = diff(y,v)

```

```

140     if dim >= 3:
141         dxdu = diff(x,w)
142         dydu = diff(y,w)
143         dzdu = diff(z,u)
144         dzdv = diff(z,v)
145         dzdw = diff(z,w)
146     if dim == 4:
147         dxdt = diff(x,t2)
148         dydt = diff(y,t2)
149         dzdt = diff(z,t2)
150         dtdu = diff(t1,u)
151         dt dv = diff(t1,v)
152         dt dw = diff(t1,w)
153         dt dt = diff(t1,t2)
154
155
156     import numpy as np
157     from sympy import Matrix
158     g = Matrix(np.zeros([dim,dim]))
159     g[0,0] = dxdu*dxdu + dydu*dydu
160     g[0,1] = dxdu*dxdv + dydu*dydv
161     g[1,1] = dxdv*dxdv + dydv*dydv
162     g[1,0] = g[0,1]
163     if dim >= 3:
164         g[0,0] += dzdu*dzdu
165         g[0,1] += dzdu*dzdv; g[1,0] = g[0,1]
166         g[0,2] = dxdu*dxdw + dydu*dydw + dzdu*dzdw; g[2,0] = g[0,2]
167         g[1,1] += dzdv*dzdv
168         g[1,2] = dxdv*dxdw + dydv*dydw + dzdv*dzdw; g[2,1] = g[1,2]
169         g[2,2] = dxdw*dxdw + dydw*dydw + dzdw*dzdw
170     if dim == 4:
171         g[0,0] += dtdu*dtdu
172         g[0,1] += dtdu*dtdv; g[1,0] = g[0,1]
173         g[0,2] += dtdu*dtdw; g[2,0] = g[0,2]
174         g[0,3] = dxdu*dxdt + dydu*dydt + dzdu*dzdt + dtdu*dtdt; g[3,0] = g[0,3]
175         g[1,1] += dt dv*dt dv
176         g[1,2] += dt dv*dt dw; g[2,1] = g[1,2]
177         g[1,3] = dx dv*dx dt + dy dv*dy dt + dz dv*dz dt + dt dv*dt dt; g[3,1] = g[1,3]
178         g[2,2] += dt dw*dt dw
179         g[2,3] = dx dw*dx dt + dy dw*dy dt + dz dw*dz dt + dt dw*dt dt; g[3,2] = g[2,3]
180         g[3,3] = dx dt*dt dt + dy dt*dy dt + dz dt*dz dt + dt dt*dt dt
181
182     if form=="simplified":
183         def simplify_expr(expr):
184             new_expr = sym.trigsimp(expr)
185             new_expr = sym.simplify(new_expr)
186             return new_expr
187         print "Performing simplification on the metric. This may take some time ...."
188         for i in range(0,dim):
189             for j in range(0,dim):
190                 g[i,j] = simplify_expr(g[i,j])
191
192     if write_to_file:
193         f = open("metric.txt","w")
194         f.write(str(g))
195         f.close()
196     return g
197
198 def toroidal_coordinates(form="simplified"):
199     from sympy.abc import u, v, w, a
200     from sympy import sin,cos,sinh,cosh
201
202     x = (a*sinh(u)*cos(w))/(cosh(u) - cos(v))
203     y = (a*sinh(u)*sin(w))/(cosh(u) - cos(v))
204     z = (a*sin(v))/(cosh(u) - cos(v))
205     coord1 = (x,y,z)
206     coord2 = (u,v,w)
207     g = metric(coord1,coord2,form)

```

```

208     diff_form = [['du*du', 'du*dv', 'du*dw'],
209                  ['dv*du', 'dv*dv', 'dv*dw'],
210                  ['dw*du', 'dv*dw', 'dw*dw']]
211     return g, diff_form
212
213 def cylindrical_coordinates(form="simplified"):
214     from sympy.abc import u, v, w, x, y, z
215     from sympy import sin, cos
216
217     x = u*cos(v)
218     y = u*sin(v)
219     z = w
220     coord1 = (x, y, z)
221     coord2 = (u, v, w)
222     g = metric(coord1, coord2, form)
223     diff_form = [['du*du', 'du*dv', 'du*dw'],
224                  ['dv*du', 'dv*dv', 'dv*dw'],
225                  ['dw*du', 'dv*dw', 'dw*dw']]
226     return g, diff_form
227
228 def spherical_coordinates(form="simplified"):
229     from sympy.abc import u, v, w, x, y, z
230     from sympy import sin, cos
231
232     x = u*sin(v)*cos(w)
233     y = u*sin(v)*sin(w)
234     z = u*cos(v)
235     coord1 = (x, y, z)
236     coord2 = (u, v, w)
237     g = metric(coord1, coord2, form)
238     diff_form = [['du*du', 'du*dv', 'du*dw'],
239                  ['dv*du', 'dv*dv', 'dv*dw'],
240                  ['dw*du', 'dv*dw', 'dw*dw']]
241     return g, diff_form
242
243 def inverse_prolate_spheroidal_coordinates(form="usimp", write_to_file=True):
244     from sympy.abc import u, v, w, a
245     from sympy import sin, cos, sinh, cosh
246
247     x = (a*sinh(u)*sin(v)*cos(w))/(cosh(u)**2 - sin(v)**2)
248     y = (a*sinh(u)*sin(v)*sin(w))/(cosh(u)**2 - sin(v)**2)
249     z = (a*cosh(u)*cos(v))/(cosh(u)**2 - sin(v)**2)
250     coord1 = (x, y, z)
251     coord2 = (u, v, w)
252     g = metric(coord1, coord2, form, write_to_file)
253     diff_form = [['du*du', 'du*dv', 'du*dw'],
254                  ['dv*du', 'dv*dv', 'dv*dw'],
255                  ['dw*du', 'dv*dw', 'dw*dw']]
256     return g, diff_form
257
258 def cylindrical_catenaoid_coordinates(form='simplified'):
259     from sympy.abc import u, v, w
260     from sympy import sin, cos
261     x = cos(u) - v*sin(u)
262     y = sin(u) + v*cos(u)
263     z = v
264     coord1 = (x, y, z)
265     coord2 = (u, v, w)
266     g = metric(coord1, coord2, form)
267     g = g[:2, :2] # 2-dimensional
268     diff_form = [['du*du', 'du*dv'],
269                  ['dv*du', 'dv*dv']]
270     return g, diff_form
271
272 def egg_carton_coordinates(form='simplified'):
273     from sympy.abc import u, v, w
274     from sympy import sin, cos
275     x = u

```

```

276 y = v
277 z = sin(u)*cos(v)
278 coord1 = (x,y,z)
279 coord2 = (u,v,w)
280 g = metric(coord1,coord2,form)
281 g = g[:2,:2] # 2-dimensional
282 diff_form = [['du*du','du*dv'],
283              ['dv*du','dv*dv']]
284 return g, diff_form
285
286 def analytical(k_value=0,form="simplified"): # k=0 gives flat space
287     from sympy import sin
288     from sympy.abc import u,v,k
289     ds2 = '(1/(1 - k*u**2))*du**2 + u**2*dv**2 + u**2*sin(v)**2*dw**2'
290     g = curve_to_metric(ds2,3)
291     g = g.subs(k,k_value)
292     diff_form = [['du*du','du*dv','du*dw'],
293                 ['dv*du','dv*dv','dv*dw'],
294                 ['dw*du','dw*dv','dw*dw']]
295     return g, diff_form
296
297 def kerr_metric(): #in polar coordinates u,v,w, and t
298     from sympy import symbols, simplify, cos, sin
299     from sympy.abc import G,M,l,u,v,w #,c,J
300     # from wikipedia :
301     """
302     ds2 = '(1 - us*u/p)*c**2*dt**2 - (p/l)*du**2 - p*dv**2 -\
303           (u**2 + a**2 + (us*u*a**2/p**2)*sin(v)**2)*sin(v)**2*dw**2\
304           + (2*us*u*a*sin(v)**2/p)*c*dt*dw'
305     g = curve_to_metric(ds2,dim=4)
306
307     us,p,a,l = symbols('us,p,a,l')
308     g = g.subs({p:u**2 + a**2*cos(v)})
309     g = g.subs({l:u**2 - us*u + a**2})
310     g = g.subs({us:2*G*M/c**2})
311     g = g.subs({a:J/(M*c)})
312     """
313     # from Thomas A. Moore (if a=0 ds2 reduces to Schwarzschild solution)
314     ds2 = '-(1 - us*u/p)*dt**2 + (p/l)*du**2 + p*dv**2 \
315           + (u**2 + a**2 + (us*u*a**2*sin(v)**2/p**2))*sin(v)**2*dw**2\
316           - (2*us*u*a*sin(v)**2/p)*dt*dw'
317     g = curve_to_metric(ds2,dim=4)
318     us,p,a,l = symbols('us,p,a,l')
319     g = g.subs({p:u**2 + a**2*cos(v)})
320     g = g.subs({l:u**2 - us*u + a**2})
321     g = g.subs({us:2*G*M})
322     print "Performing simplification on the metric. This may take some time ...."
323     g = simplify(g)
324     diff_form = [['du*du','du*dv','du*dw','dt*du'],
325                 ['dv*du','dv*dv','dv*dw','dv*dt'],
326                 ['dw*du','dw*dv','dw*dw','dw*dt'],
327                 ['dt*du','dt*dv','dt*dw','dt*dt']]
328     return g, diff_form
329
330 def kerr_3D_metric_time_independent(): # unphysical ?
331     from sympy import symbols, simplify, cos, sin
332     from sympy.abc import G,M,l,u,v,w
333     ds2 = '(p/l)*du**2 + p*dv**2 \
334           + (u**2 + a**2 + (us*u*a**2*sin(v)**2/p**2))*sin(v)**2*dw**2'
335     g = curve_to_metric(ds2,dim=3)
336     us,p,a,l = symbols('us,p,a,l')
337     g = g.subs({p:u**2 + a**2*cos(v)})
338     g = g.subs({l:u**2 - us*u + a**2})
339     g = g.subs({us:2*G*M})
340     print "Performing simplification on the metric. This may take some time ...."
341     g = simplify(g)
342     diff_form = [['du*du','du*dv','du*dw'],
343                 ['dv*du','dv*dv','dv*dw'],

```

```

344         ['dw*du', 'dv*dw', 'dw*dw']]
345     return g, diff_form
346
347 def kerr_3D_metric(): # one space component dropped : phi
348     from sympy import symbols, simplify, cos, sin
349     from sympy.abc import G, M, l, u, v, t
350     ds2 = '-(1 - us*u/p)*dw**2 + (p/l)*du**2 + p*dv**2'
351     g = curve_to_metric(ds2, dim=3)
352     us, p, a, l = symbols('us, p, a, l')
353     g = g.subs({p: u**2 + a**2*cos(v)})
354     g = g.subs({l: u**2 - us*u + a**2})
355     g = g.subs({us: 2*M})
356     print "Performing simplification on the metric. This may take some time ...."
357     g = simplify(g)
358     diff_form = [['du*du', 'du*dv', 'du*dw'],
359                 ['dv*du', 'dv*dv', 'dv*dw'],
360                 ['dw*du', 'dv*dw', 'dw*dw']]
361     return g, diff_form
362
363 def torus_metric(a=1, c=2, form='simplified'):
364     from sympy.abc import u, v, w
365     from sympy import sin, cos
366     x = (c + a*cos(u))*cos(v)
367     y = (c + a*cos(u))*sin(v)
368     z = sin(u)
369     coord1 = (x, y, z)
370     coord2 = (u, v, w)
371     g = metric(coord1, coord2, form)
372     g = g[:2, :2]
373     diff_form = [['du*du', 'du*dv'],
374                 ['dv*du', 'dv*dv']]
375     return g, diff_form
376
377 def flat_sphere():
378     ds2 = 'dv**2 + sin(v)**2*dw**2'
379     diff_form = [['dv*dv', 'dv*dw'], ['dw*dv', 'dw*dw']]
380     g = curve_to_metric(ds2, dim=2, diff_=diff_form)
381     return g, diff_form
382
383 def mobius_strip(form='simplified'):
384     from sympy.abc import u, v, w
385     from sympy import sin, cos
386     x = (1 + cos(u/2)*v/2)*cos(u)
387     y = (1 + cos(u/2)*v/2)*sin(u)
388     z = sin(u/2)*v/2
389     coord1 = (x, y, z)
390     coord2 = (u, v, w)
391     g = metric(coord1, coord2, form)
392     g = g[:2, :2]
393     diff_form = [['du*du', 'du*dv'], ['dv*du', 'dv*dv']]
394     return g, diff_form
395
396 if __name__ == "__main__":
397     g1, diff_form = toroidal_coordinates()
398     print "The toroidal metric"
399     print g1
400     print 'with the corresponding differentials'
401     print diff_form
402
403     """
404     g2, diff_form = cylindrical_coordinates()
405     print "\nCylindrical"
406     print g2
407
408     g3, diff_form = spherical_coordinates()
409     print "\nSpherical"
410     print g3
411

```

```

412 g4, diff_form = inverse_prolate_spheroidal_coordinates("usimp",1)
413 print "\nInverse prolate spheroidal coordinates - without simplified form"
414 print g4
415 """

```

C.4 Riemann Curvature, Ricci tensor, Scalar curvature

tensor.py

```

1 from __future__ import division
2 import sympy as sympy
3
4 class Riemann:
5     """
6     Used for defining a Riemann curvature tensor or Ricci tensor
7     for a given metric between cartesian coordinates and another
8     orthogonal coordinate system.
9     """
10    def __init__(self, g, dim, sys_title="coordinate_system", user_coord = None,
11                 flat_diff = None):
12        """
13        Contructor __init__ initializes the object for a given
14        metric g (symbolic matrix). The metric must be defined
15        with sympy variables u and v for the orthogonal basis in
16        the Cartesian coordinate system.
17
18        g : metric defined as nxn Sympy.Matrix object
19        sys_titles : descriptive information about the coordinate system
20                   besides Cartesian coordinates
21        dim : R^2, R^3, or R^4
22        user_coord : User supplies their own set of coordinate symbols
23        flat_diff : Matrix with differentials if g is a flat metric
24        """
25    from sympy.diffgeom import Manifold, Patch
26    self.dim = dim
27    self.g = g
28    if flat_diff is not None:
29        self._set_flat_coordinates(sys_title, flat_diff)
30    elif user_coord is None:
31        self._set_coordinates(sys_title)
32    else:
33        self._set_user_coordinates(sys_title, user_coord)
34    self.metric = self._metric_to_twoform(g)
35
36    def _set_flat_coordinates(self, sys_title, flat_diff):
37    from sympy.diffgeom import CoordSystem, Manifold, Patch
38    manifold = Manifold("M", self.dim)
39    patch = Patch("P", manifold)
40    flat_diff = sympy.Matrix(flat_diff)
41    N = flat_diff.shape[0]
42    coords = []
43    for i in range(0, N):
44        n = str(flat_diff[i, i]).find('*')
45        coord_i = str(flat_diff[i, i])[1:n]
46        coords.append(coord_i)
47    if self.dim==4:
48        system = CoordSystem(sys_title, patch, [str(coords[0]), str(coords[1]), \
49                                                str(coords[2]), str(coords[3])])
50        u, v, w, t = system.coord_functions()
51        self.w = w
52        self.t = t
53
54    if self.dim==3:
55        system = CoordSystem(sys_title, patch, [str(coords[0]), str(coords[1]), \
56                                                str(coords[2])])
57        u, v, w = system.coord_functions()
58        self.w = w

```

```

59
60     if self.dim==2:
61         system = CoordSystem(sys_title, patch, [str(coords[0]),str(coords[1])])
62         u, v = system.coord_functions()
63
64     self.u, self.v = u, v
65     self.system = system
66
67     def _set_user_coordinates(self,sys_title,user_coord):
68         from sympy.diffgeom import CoordSystem, Manifold, Patch
69         manifold = Manifold("M",self.dim)
70         patch = Patch("P",manifold)
71         if self.dim==4:
72             system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1]),\
73                                                     str(user_coord[2]),str(user_coord[3])])
74             u, v, w, t = system.coord_functions()
75             self.w = w
76             self.t = t
77
78         if self.dim==3:
79             system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1]),
80                                                     str(user_coord[2])])
81             u, v, w = system.coord_functions()
82             self.w = w
83
84         if self.dim==2:
85             system = CoordSystem(sys_title, patch, [str(user_coord[0]),str(user_coord[1])])
86             u, v = system.coord_functions()
87
88     self.u, self.v = u, v
89     self.system = system
90
91     def _set_coordinates(self,sys_title):
92         from sympy.diffgeom import CoordSystem, Manifold, Patch
93         manifold = Manifold("M",self.dim)
94         patch = Patch("P",manifold)
95         if self.dim==4:
96             system = CoordSystem(sys_title, patch, ["u", "v", "w","t"])
97             u, v, w, t = system.coord_functions()
98             self.w = w
99             self.t = t
100
101         if self.dim==3:
102             system = CoordSystem(sys_title, patch, ["u", "v", "w"])
103             u, v, w = system.coord_functions()
104             self.w = w
105
106         if self.dim==2:
107             system = CoordSystem(sys_title, patch, ["u", "v"])
108             u, v = system.coord_functions()
109
110     self.u, self.v = u, v
111     self.system = system
112
113     def _metric_to_twoform(self,g):
114         dim = self.dim
115         system = self.system
116         diff_forms = system.base_oneforms()
117         u_, v_ = self.u, self.v
118         u = u_
119         v = v_
120         if dim >= 3:
121             w_ = self.w
122             w = w_
123         if dim == 4:
124             t_ = self.t
125             t = t_
126

```



```

127 from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
128 import sympy.abc as abc
129 self._abc = abc
130 self._symbols = ['*', '/', '(', ')', '"', "'", '']
131 self._letters = []
132 g_ = sympy.Matrix(dim*[dim*[0]])
133 # re-evaluate the metric for (u,v,w,t if 4D) which are Basescalar objects
134 for i in range(dim):
135     for j in range(dim):
136         expr = str(g[i,j])
137         self._try_expr(expr) # evaluate expr in a safe environment
138         for letter in self._letters:
139             exec('from sympy.abc import %s'%letter)
140         g_[i,j] = eval(expr) # this will now work for any variables defined in sympy.abc
141         g_[i,j] = g_[i,j].subs(u,u_)
142         g_[i,j] = g_[i,j].subs(v,v_)
143         if dim >= 3:
144             g_[i,j] = g_[i,j].subs(w,w_)
145         if dim == 4:
146             g_[i,j] = g_[i,j].subs(t,t_)
147 from sympy.diffgeom import TensorProduct
148 metric_diff_form = sum([TensorProduct(di, dj)*g_[i, j]
149                          for i, di in enumerate(diff_forms)
150                          for j, dj in enumerate(diff_forms)])
151 return metric_diff_form
152
153 def _try_expr(self,expr):
154     """
155     This is a help function used initially to evaluate the user-defined metric
156     elements as a sympy expression : expr. The purpose of this method is to
157     prevent the namespace of the user from being polluted by the command
158     'from sympy.abc import *'.
159
160     expr : a string object to be evaluated as a sympy expression
161     """
162 from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
163 letters = self._letters
164 abc = self._abc
165 try:
166     for letter in letters:
167         exec('from sympy.abc import %s'%letter) # re-execute after finding each unknown variable
168     l_ = expr.count('(')
169     r_ = expr.count(')')
170     if l_ == r_:
171         eval(expr)
172     elif l_ < r_:
173         eval((r_-l_)*'('+expr)
174 except NameError as err:
175     msg = str(err)
176     pos = msg.find('"')
177     letter = msg[pos+1]
178     pos = pos + 1
179     found = False
180     symbols = self._symbols
181     while (pos+1 < len(msg)) and (not found):
182         more = msg[pos+1]
183         for symb in symbols:
184             if more==symb or more.isdigit():
185                 found = True
186                 break
187         if found is False:
188             letter = letter+more
189             pos = pos + 1
190 for alphabet in abc.__dict__:
191     if letter == alphabet:
192         letters.append(alphabet)
193         self._try_expr(expr[expr.find(alphabet):]) # search for the next unknown variable
194

```

```

195 def _tuple_to_list(self,t):
196     """
197     Recursively turn a tuple to a list.
198     """
199     return list(map(self._tuple_to_list, t)) if isinstance(t, (list, tuple)) else t
200
201 def _simplify_expr(self,expr): # this is a costly stage for complex expressions
202     """
203     Perform simplification of the provided expression.
204     Method returns a SymPy expression.
205     """
206     expr = sympy.trigsimp(expr)
207     expr = sympy.simplify(expr)
208     return expr
209
210 def metric_to_Christoffel_1st(self):
211     from sympy.diffgeom import metric_to_Christoffel_1st
212     return metric_to_Christoffel_1st(self.metric)
213
214 def metric_to_Christoffel_2nd(self):
215     from sympy.diffgeom import metric_to_Christoffel_2nd
216     return metric_to_Christoffel_2nd(self.metric)
217
218 def find_Christoffel_tensor(self,form="simplified"):
219     """
220     Method determines the Riemann-Christoffel tensor
221     for a given metric(which must be in two-form).
222
223     form : default value - "simplified"
224     If desired, a simplified form is returned.
225
226     The returned value is a SymPy Matrix.
227     """
228     from sympy.diffgeom import metric_to_Riemann_components
229     metric = self.metric
230     R = metric_to_Riemann_components(metric)
231     simpR = self._tuple_to_list(R)
232     dim = self.dim
233     if form=="simplified":
234         print 'Performing simplifications on each component....'
235         for m in range(dim):
236             for i in range(dim):
237                 for j in range(dim):
238                     for k in range(dim):
239                         expr = str(R[m][i][j][k])
240                         expr = self._simplify_expr(expr)
241                         simpR[m][i][j][k] = expr
242     self.Christoffel = sympy.Matrix(simpR)
243     return self.Christoffel
244
245 def find_Ricci_tensor(self,form="simplified"):
246     """
247     Method determines the Ricci curvature tensor for
248     a given metric(which must be in two-form).
249
250     form : default value - "simplified"
251     If desired, a simplified form is returned.
252
253     The returned value is a SymPy Matrix.
254     """
255     from sympy.diffgeom import metric_to_Ricci_components
256     metric = self.metric
257     RR = metric_to_Ricci_components(metric)
258     simpRR = self._tuple_to_list(RR)
259     dim = self.dim
260     if form=="simplified":
261         print 'Performing simplifications on each component....'
262         for m in range(dim):

```

```

263         for i in range(dim):
264             expr = str(RR[m][i])
265             expr = self._simplify_expr(expr)
266             simpRR[m][i] = expr
267         self.Ricci = sympy.Matrix(simpRR)
268         return self.Ricci
269
270     def find_scalar_curvature(self):
271         """
272         Method performs scalar contraction on the Ricci tensor.
273         """
274         try:
275             Ricci = self.Ricci
276         except AttributeError:
277             print "Ricci tensor must be determined first."
278             return None
279         g = self.g
280         g_inv = self.g.inv()
281         scalar_curv = sympy.simplify(g_inv*Ricci)
282         scalar_curv = sympy.trace(scalar_curv)
283         self.scalar_curv = scalar_curv
284         return self.scalar_curv
285
286
287 if __name__ == "__main__":
288     import find_metric
289     k = -1
290     g,diff_form = find_metric.analytical(k) # k=0 gives flat space
291     R = Riemann(g,dim=3,sys_title="analytical")
292     print R.metric
293     from sympy import srepr
294     print srepr(R.system)
295     RC = R.find_Christoffel_tensor()
296     RR = R.find_Ricci_tensor()
297     scalarRR = R.find_scalar_curvature()
298
299     print "\nThe analytical curve element has the following metric for k=%.1f"%k
300     print g
301     print "\nThe Ricci tensor is given as"
302     print RR
303     print "\nand the scalar curvature is"
304     print scalarRR
305
306     """
307     from sympy.abc import r,theta, phi, u,v
308     g,diff_form = find_metric.flat_sphere()
309     diff = [['dv*dv','dv*dw'],['dw*dv','dw*dw']]
310     R = Riemann(g, dim=2, sys_title="flat_sphere",\
311                flat_metric = True, flat_diff = diff)
312     C = R.metric_to_Christoffel_2nd(R.metric)
313     RC = R.find_Christoffel_tensor()
314     RR = R.find_Ricci_tensor()
315     scalarRR = R.find_scalar_curvature()
316
317     print "\nThe 2D sphere has the following metric"
318     print g
319     print "\nThe Christoffel tensor is given as"
320     for m in range(dim):
321         for i in range(dim):
322             print RC[m,i]
323     print "\nThe Ricci tensor is given as"
324     print RR
325     print "\nand the scalar curvature is"
326     print scalarRR
327
328
329     g,diff_form = find_metric.toroidal_coordinates()
330     R = Riemann(g,dim=3,sys_title="toroidal")

```

```

331     RC = R.find_Christoffel_tensor()
332     RR = R.find_Ricci_tensor()
333     print RC,"\n",RR
334
335     g,diff_form = find_metric.spherical_coordinates()
336     R = Riemann(g,dim=3,sys_title="spherical")
337     RC = R.find_Christoffel_tensor()
338     RR = R.find_Ricci_tensor()
339     print RC,"\n",RR
340
341     g,diff_form = find_metric.cylindrical_coordinates()
342     R = Riemann(g=g,dim=3,sys_title="cylindrical")
343     RC = R.find_Christoffel_tensor()
344     RR = R.find_Ricci_tensor()
345     print RC,"\n",RR
346
347     # Warning : This takes very long time (just to find g)!
348     g,diff_form = find_metric.inverse_prolate_spheroidal_coordinates()
349     R = Riemann(g,dim=3,sys_title="inv_prolate_sphere")
350     RC = R.find_Christoffel_tensor()
351     RR = R.find_Ricci_tensor()
352     print RC,"\n",RR
353     """

```

C.5 Geodesic Differential Equations Solver

gde.py

```

1  import numpy as np
2  import scipy.integrate as sc
3  import sympy as sym
4
5  def f3D(y,s,*args):
6      C,u,v,w = args
7      y0 = y[0] # u
8      y1 = y[1] # u'
9      y2 = y[2] # v
10     y3 = y[3] # v'
11     y4 = y[4] # w
12     y5 = y[5] # w'
13     C = C.subs({u:y0,v:y2,w:y4})
14
15     dy = np.zeros_like(y)
16     dy[0] = y1
17     dy[2] = y3
18     dy[4] = y5
19     dy[1] = -C[0,0][0]*dy[0]**2\
20             -2*C[0,0][1]*dy[0]*dy[2]\
21             -2*C[0,0][2]*dy[0]*dy[4]\
22             -2*C[0,1][2]*dy[2]*dy[4]\
23             -C[0,1][1]*dy[2]**2\
24             -C[0,2][2]*dy[4]**2
25     dy[3] = -C[1,0][0]*dy[0]**2\
26             -2*C[1,0][1]*dy[0]*dy[2]\
27             -2*C[1,0][2]*dy[0]*dy[4]\
28             -2*C[1,1][2]*dy[2]*dy[4]\
29             -C[1,1][1]*dy[2]**2\
30             -C[1,2][2]*dy[4]**2
31     dy[5] = -C[2,0][0]*dy[0]**2\
32             -2*C[2,0][1]*dy[0]*dy[2]\
33             -2*C[2,0][2]*dy[0]*dy[4]\
34             -2*C[2,1][2]*dy[2]*dy[4]\
35             -C[2,1][1]*dy[2]**2\
36             -C[2,2][2]*dy[4]**2
37     return dy
38
39 def f(y,s,*args):

```

```

40 """
41 The geodesic differential equations are solved.
42 Described as a system of first order differential-
43 equations :
44
45 y0 = u
46 y1 = u'
47 y2 = v
48 y3 = v'
49
50 dy0 = y1
51 dy1 = u'',
52 dy2 = y2
53 dy3 = v''
54
55 Input :
56 C is the Christoffel symbol of second kind
57 u and v are symbolic expressions.
58
59 Output :
60 dy = [dy0,dy1,dy2,dy3]
61 """
62 C,u,v = args
63 y0 = y[0] # u
64 y1 = y[1] # u'
65 y2 = y[2] # v
66 y3 = y[3] # v'
67 dy = np.zeros_like(y)
68 dy[0] = y1
69 dy[2] = y3
70
71 C = C.subs({u:y0,v:y2})
72 dy[1] = -C[0,0][0]*dy[0]**2\
73         -2*C[0,0][1]*dy[0]*dy[2]\
74         -C[0,1][1]*dy[2]**2
75 dy[3] = -C[1,0][0]*dy[0]**2\
76         -2*C[1,0][1]*dy[0]*dy[2]\
77         -C[1,1][1]*dy[2]**2
78 return dy
79
80 def solve(C,u0,s0,s1,ds,solver=None):
81     from sympy.abc import u,v
82     global f
83     if len(u0) == 6: # 3D problem
84         from sympy.abc import w
85         args = (C,u,v,w)
86         f = f3D
87     else:
88         args = (C,u,v)
89
90     if solver == None: # use lsoda from scipy.integrate.odeint
91         s = np.arange(s0,s1+ds,ds)
92         print 'Running solver ...'
93         return sc.odeint(f,u0,s,args=args)
94     else: # use any other solver from scipy.integrate.ode
95         # vode,zvode,lsoda,dopri5,dop853
96         r = sc.ode(lambda t,x,args: f(x,t,*args)).set_integrator(solver)
97         r.set_f_params(args)
98         r.set_initial_value(u0)
99         y = []
100        print 'Running solver ...'
101        while r.successful() and r.t <= s1:
102            r.integrate(r.t + ds)
103            y.append(r.y)
104        return np.array(y)
105
106 def two_points(p1,p2,s0,s1,ds,C,tol=1e-6,surface=None):
107     """

```

```

108 The function attempts to find the geodesic between two points p1 and p2.
109 """
110 p1 = np.array(p1)
111 p2 = np.array(p2)
112 if (np.fabs(p1-p2) <= tol).all() == 1:
113     raise ValueError('Point 1 and point 2 are the same point : (%.1f,%.1f)%(p2[0],p2[1]))
114 found = False
115 X_ = []
116 u_ = 4*[0]; u_[0] = p1[0]; u_[2] = p1[1]
117 du = np.arange(-.2,.2,ds)
118 N = du.shape[0]
119 i = 0
120 while (i < N) and (not found):
121     u_[1] = du[i]
122     j = 0
123     while (j < N) and (not found):
124         u_[3] = du[j]
125         print 'Testing initial conditions : '
126         print u_
127         X = solve(C,u_,s0,s1,ds)
128         u__ = np.where(np.fabs(X[:,0]-p2[0]) <= tol)[0]
129         v__ = np.where(np.fabs(X[:,2]-p2[1]) <= tol)[0]
130         if (u__ == v__).any() == True:
131             found = True
132             X_ = X
133             print 'Following initial conditions connect the two provided points '
134             print '(%f,%f)%(u_[0],u_[2]), ', ' (%f,%f)%(p2[0],p2[1])
135             print "u' = %f , v' = %f"%(u_[1],u_[3])
136         j = j + 1
137     i = i + 1
138 if (len(X_) > 0) and (surface is not None):
139     print 'Plotting the geodesics for provided surface...',surface
140     if surface == 'catenoid':
141         display_catenoid(u_,s0,s1,ds,show=True)
142     elif surface == 'torus':
143         display_torus(u_,s0,s1,ds,show=True)
144     elif surface == 'sphere':
145         display_sphere(u_,s0,s1,ds,show=True)
146     elif surface == 'egg_carton':
147         display_egg_carton(u_,s0,s1,ds,show=True)
148 return X
149
150 def Christoffel_2nd(g=None,metric=None): # either g is supplied as arugment or the two-form
151     from sympy.abc import u,v
152     from sympy.diffgeom import metric_to_Christoffel_2nd
153     from sympy import asin, acos, atan, cos, log, ln, exp, cosh, sin, sinh, sqrt, tan, tanh
154     if metric is None: # if metric is not specified as two_form
155         import tensor as t
156         R = t.Riemann(g,g.shape[0])
157         metric = R.metric
158     C = sym.Matrix(eval(str(metric_to_Christoffel_2nd(metric))))
159     return C
160
161 def catenoid():
162     import find_metric
163     g = find_metric.cylindrical_catenoid_coordinates()
164     C = Christoffel_2nd(g)
165     return C
166
167 def torus(a=1,c=2):
168     import find_metric
169     g = find_metric.torus_metric(a,c)
170     C = Christoffel_2nd(g)
171     return C
172
173 def toroid(u=1,v=None,a=1):
174     import find_metric as fm
175     g, diff = fm.toroidal_coordinates()

```

```

176     if v is None:
177         g = g.subs('u',u)[:2,:2]
178     else:
179         g = g.subs('v',v)[1:,1:]
180     g = g.subs('a',a)
181
182     import tensor as t
183     R = t.Riemann(g,dim=2,sys_title='toroid')
184     C = Christoffel_2nd(metric=R.metric)
185     return C
186
187
188 def egg_carton():
189     import tensor as t
190     import find_metric as fm
191     g,diff = find_metric.egg_carton_metric()
192     R = t.Riemann(g,dim=2,sys_title='egg_carton',flat_diff=diff)
193     """
194     # this works :
195     from sympy.abc import u,v
196     u_,v_ = R.system.coord_functions()
197     du,dv = R.system.base_oneforms()
198     metric = R.metric.subs({u:u_,v:v_,'dv':dv,'du':du})
199     """
200     C = Christoffel_2nd(metric=R.metric)
201     return C
202
203 def flat_kerr(a=0,G=1,M=0.5):
204     import find_metric as fm
205     from sympy.diffgeom import CoordSystem, Manifold, Patch, TensorProduct
206
207     manifold = Manifold("M",3)
208     patch = Patch("P",manifold)
209     kerr = CoordSystem("kerr", patch, ["u","v","w"])
210     u,v,w = kerr.coord_functions()
211     du,dv,dw = kerr.base_oneforms()
212
213     g11 = (a**2*sym.cos(v) + u**2)/(-2*G*M*u + a**2 + u**2)
214     g22 = a**2*sym.cos(v) + u**2
215     g33 = -(1 - 2*G*M*u/(u**2 + a**2*sym.cos(v)))
216     # time independent : unphysical ?
217     #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
218     #**2
219     metric = g11*TensorProduct(du, du) + g22*TensorProduct(dv, dv) + g33*TensorProduct(dw, dw)
220     C = Christoffel_2nd(metric=metric)
221     return C
222
223 def flat_sphere():
224     import find_metric as fm
225     import tensor as t
226     g,diff = find_metric.flat_sphere()
227     R = t.Riemann(g,dim=2,sys_title='flat_sphere',flat_diff=diff)
228     C = Christoffel_2nd(metric=R.metric)
229     return C
230
231 def sphere():
232     from sympy.abc import u,v
233     from sympy import tan, cos ,sin
234     """
235     return flat_sphere() # in correct entries in Christoffel symbol of 2nd kind
236     """
237     return sym.Matrix([[0,-tan(v)), (0, 0)],[(sin(v)*cos(v), 0), (0, 0)]]
238
239 def mobius_strip():
240     import find_metric as fm
241     import tensor as t
242     g,diff = fm.mobius_strip()
243     R = t.Riemann(g,dim=2,sys_title='mobius_strip',flat_diff = diff)

```

```

243 #metric=R.metric
244 from sympy.diffgeom import TensorProduct, Manifold, Patch, CoordSystem
245 manifold = Manifold("M",2)
246 patch = Patch("P",manifold)
247 system = CoordSystem('mobius_strip', patch, ["u", "v"])
248 u, v = system.coord_functions()
249 du,dv = system.base_oneforms()
250 from sympy import cos
251 metric = (cos(u/2)**2*v**2/4 + cos(u/2)*v + v**2/16 + 1)*TensorProduct(du, du) + 0.25*TensorProduct(
    dv, dv)
252 C = Christoffel_2nd(metric=metric)
253 return C
254
255 def display_mobius_strip(u0,s0,s1,ds,solver=None,show=False):
256     C = mobius_strip() # Find the Christoffel tensor for mobius strip
257     X = solve(C,u0,s0,s1,ds,solver)
258
259     import matplotlib.pyplot as plt
260     from mpl_toolkits.mplot3d import Axes3D
261     u,v = plt.meshgrid(np.linspace(-2*np.pi,np.pi,250),np.linspace(-np.pi,np.pi,250))
262     x = (1 + np.cos(u/2.)*v/2.)*np.cos(u)
263     y = (1 + np.cos(u/2.)*v/2.)*np.sin(u)
264     z = np.sin(u/2.)*v/2.
265
266     fig = plt.figure()
267     ax = fig.add_subplot(111, projection='3d')
268     ax.view_init(elev=10, azim=81)
269     # use transparent colormap
270     import matplotlib.cm as cm
271     theCM = cm.get_cmap()
272     theCM._init()
273     alphas = -.5*np.ones(theCM.N)
274     theCM._lut[:,-3,-1] = alphas
275     ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
276     ax.set_xlabel('x')
277     ax.set_ylabel('y')
278     ax.set_zlabel('z')
279     plt.hold('on')
280
281     # plot the parametrized data on to the catenoid
282     u,v = X[:,0], X[:,2]
283     x = (1 + np.cos(u/2.)*v/2.)*np.cos(u)
284     y = (1 + np.cos(u/2.)*v/2.)*np.sin(u)
285     z = np.sin(u/2.)*v/2.
286
287     ax.plot(x,y,z,'--r')
288     s0_ = s0/np.pi
289     s1_ = s1/np.pi
290     fig.suptitle("$s$ in [%1.f,%1.f$\\pi$] $ , $u$ = %1.f$ , $u$' = %1.f$ , $v$ = %1.f$ , $v$' = %1.f$"%(s0,s1_,u0
        [0],u0[1],u0[2],u0[3]))
291     if show == True:
292         plt.show()
293     return X,plt
294
295 def display_catenoid(u0,s0,s1,ds,solver=None,show=False):
296     C = catenoid() # Find the Christoffel tensor for cylindrical catenoid
297     X = solve(C,u0,s0,s1,ds,solver)
298
299     import matplotlib.pyplot as plt
300     from mpl_toolkits.mplot3d import Axes3D
301     N = X[:,0].shape[0]
302     u,v = plt.meshgrid(np.linspace(-np.pi,np.pi,150),np.linspace(-np.pi,np.pi,150))
303     x = np.cos(u) - v*np.sin(u)
304     y = np.sin(u) + v*np.cos(u)
305     z = v
306
307     fig = plt.figure()
308     ax = fig.add_subplot(111, projection='3d')

```



```

309     ax.view_init(elev=20, azim=-163)
310     # use transparent colormap
311     import matplotlib.cm as cm
312     theCM = cm.get_cmap()
313     theCM._init()
314     alphas = -.5*np.ones(theCM.N)
315     theCM._lut[:,-3,-1] = alphas
316     ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
317     plt.hold('on')
318
319     # plot the parametrized data on to the catenoid
320     u,v = X[:,0], X[:,2]
321     x = np.cos(u) - v*np.sin(u)
322     y = np.sin(u) + v*np.cos(u)
323     z = v
324
325     ax.plot(x,y,z,'--r')
326     s0_ = s0/np.pi
327     s1_ = s1/np.pi
328     fig.suptitle("$s$ in  $[%.1f\pi, %.1f\pi]$  , $u$ =  $%.1f\pi$  , $v$ =  $%.2f\pi$ "%(s0_,s1_,u0[1],u0[3]))
329     if show == True:
330         plt.show()
331     return X,plt
332
333 def display_sphere(u0,s0,s1,ds,solver=None,metric=None,show=False):
334     if metric == 'flat':
335         C = flat_sphere()
336         if u0[0] == 0 or u0[2] == 0:
337             print 'Division by zero may occur for provided values of u(s0) and v(s0)'
338         else:
339             C = sphere()
340     X = solve(C,u0,s0,s1,ds,solver)
341     import matplotlib.pyplot as plt
342     from mpl_toolkits.mplot3d import Axes3D
343     u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
344     x = np.cos(u)*np.cos(v)
345     y = np.sin(u)*np.cos(v)
346     z = np.sin(v)
347
348     fig = plt.figure()
349     ax = fig.add_subplot(111, projection='3d')
350     if metric == 'flat':
351         ax.view_init(elev=90., azim=0)
352     else:
353         ax.view_init(elev=0., azim=13)
354     ax.plot_surface(x,y,z,linewidth=0,cmap='Pastell1')
355     plt.hold('on')
356     # plot the parametrized data on to the sphere
357     u,v = X[:,0], X[:,2]
358     x = np.cos(u)*np.cos(v)
359     y = np.sin(u)*np.cos(v)
360     z = np.sin(v)
361
362     ax.plot(x,y,z,'--r')
363     from math import pi
364     s1_ = s1/pi
365     fig.suptitle("$s$ in  $[%.1f, %.1f\pi]$  , $u$ =  $%.1f\pi$  , $v$ =  $%.1f\pi$ "%(s0,s1_,u0[1],u0[3]))
366     if show == True:
367         plt.show()
368     return X,plt
369
370 def display_torus(u0,s0,s1,ds,a=1,c=2,solver=None,show=False):
371     C = torus(a,c) # Find the Christoffel tensor for the torus
372     X = solve(C,u0,s0,s1,ds,solver)
373
374     import matplotlib.pyplot as plt
375     from mpl_toolkits.mplot3d import Axes3D
376     N = X[:,0].shape[0]

```

```

377 u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
378 x = (c + a*np.cos(v))*np.cos(u)
379 y = (c + a*np.cos(v))*np.sin(u)
380 z = np.sin(v)
381
382 fig = plt.figure()
383 ax = fig.add_subplot(111, projection='3d')
384 ax.view_init(elev=-60, azim=100)
385 # use transparent colormap -> negative
386 import matplotlib.cm as cm
387 theCM = cm.get_cmap()
388 theCM._init()
389 alphas = 2*np.ones(theCM.N)
390 theCM._lut[:,-3,-1] = alphas
391 ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
392 plt.hold('on')
393
394 # plot the parametrized data on to the torus
395 u,v = X[:,0], X[:,2]
396 x = (c + a*np.cos(v))*np.cos(u)
397 y = (c + a*np.cos(v))*np.sin(u)
398 z = np.sin(v)
399
400 ax.plot(x,y,z,'--r')
401 s1_ = s1/pi
402 fig.suptitle("$s\\in[1.f,1.f\\pi]$ , $u = %.1f$ , $u' = %.1f$ , $v = %.1f$ , $v' = %.1f$"%(s0,s1_,u0
403 [0],u0[1],u0[2],u0[3]))
404 if show == True:
405     plt.show()
406 return X,plt
407
408 def display_toroid(u0,s0,s1,ds,u_val=1,v_val=None,a=1,solver=None,show=False):
409     C = toroid(u_val,v_val,a) # Find the Christoffel tensor for toroid
410     X = solve(C,u0,s0,s1,ds,solver)
411
412 import matplotlib.pyplot as plt
413 from mpl_toolkits.mplot3d import Axes3D
414 from math import pi
415 if v_val is None:
416     u = u_val # toroids
417     v,w = plt.meshgrid(np.linspace(-pi,pi,250),np.linspace(0,2*pi,250))
418 else:
419     v = v_val # spherical bowls
420     u,w = plt.meshgrid(np.linspace(0,2,250),np.linspace(0,2*pi,250))
421
422 x = (a*np.sinh(u)*np.cos(w))/(np.cosh(u) - np.cos(v))
423 y = (a*np.sinh(u)*np.sin(w))/(np.cosh(u) - np.cos(v))
424 z = (a*np.sin(v))/(np.cosh(u) - np.cos(v))
425
426 fig = plt.figure()
427 ax = fig.add_subplot(111, projection='3d')
428 ax.view_init(elev=90., azim=0)
429 # use transparent colormap
430 import matplotlib.cm as cm
431 theCM = cm.get_cmap()
432 theCM._init()
433 alphas = -.5*np.ones(theCM.N)
434 theCM._lut[:,-3,-1] = alphas
435 ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
436 plt.hold('on')
437
438 # plot the parametrized data on to the toroid
439 if v_val is None:
440     w,v = X[:,0], X[:,2]
441 else:
442     w,u = X[:,0], X[:,2]
443 x = (a*np.sinh(u)*np.cos(w))/(np.cosh(u) - np.cos(v))
444 y = (a*np.sinh(u)*np.sin(w))/(np.cosh(u) - np.cos(v))

```

```

444     z = (a*np.sin(v))/(np.cosh(u) - np.cos(v))
445
446     s1_ = s1/pi
447     ax.plot(x,y,z,'--r')
448     fig.suptitle('$s$ in  $[\%1f\pi \ , \ , \ , \%2.1f\pi]$  '$'(s0,s1_))
449     if show == True:
450         plt.show()
451     return X,plt
452
453 def display_egg_carton(u0,s0,s1,ds,solver=None,show=False):
454     C = egg_carton() # Find the Christoffel tensor for egg carton surface
455     X = solve(C,u0,s0,s1,ds,solver)
456
457     import matplotlib.pyplot as plt
458     from mpl_toolkits.mplot3d import Axes3D
459     from math import pi
460     N = X[:,0].shape[0]
461     u,v = plt.meshgrid(np.linspace(s0,s1,N),np.linspace(s0,s1,N))
462     x = u
463     y = v
464     z = np.sin(u)*np.cos(v)
465
466     fig = plt.figure()
467     ax = fig.add_subplot(111, projection='3d')
468     ax.view_init(elev=90., azim=0)
469     # use transparent colormap
470     import matplotlib.cm as cm
471     theCM = cm.get_cmap()
472     theCM._init()
473     alphas = -.5*np.ones(theCM.N)
474     theCM._lut[:-3,-1] = alphas
475     ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
476     plt.hold('on')
477
478     # plot the parametrized data on to the egg carton
479     u,v = X[:,0], X[:,2]
480     x = u
481     y = v
482     z = np.sin(u)*np.cos(v)
483
484     s0_ = s0/pi
485     s1_ = s1/pi
486     ax.plot(x,y,z,'--r')
487     fig.suptitle('$s$ in  $[\%1f\pi \ , \ , \ , \%2.1f\pi]$  '$'(s0_,s1_))
488     if show == True:
489         plt.show()
490     return X,plt
491
492 def display_3D_Kerr(u0,s0,s1,ds,solver=None,show=True,args=None,multiple=True):
493     if args == None:
494         C = flat_kerr() # use default values
495     else:
496         a = args[0]
497         G = args[1]
498         M = args[2]
499         C = flat_kerr(a,G,M) # Find the Christoffel tensor for 3D Kerr metric on 4D manifold
500
501     import matplotlib.pyplot as plt
502     from mpl_toolkits.mplot3d import Axes3D
503     fig = plt.figure()
504     ax = fig.add_subplot(111, projection='3d')
505     if multiple is not True:
506         X = solve(C,u0,s0,s1,ds,solver)
507         r = X[:,0]
508         theta = X[:,2]
509         # for time independent kerr metric use :
510         #phi = X[:,4]
511         #x = r*np.sin(theta)*np.cos(phi)

```

```

512     #y = r*np.sin(theta)*np.sin(phi)
513     #z = r*np.cos(theta)
514     #ax.plot(x,y,z,'b')
515     t = X[:,4]
516     x = r*np.sin(theta)
517     y = r*np.cos(theta)
518     z = t
519     ax.plot(x,y,z,'b')
520
521     if multiple is True:
522         plt.hold('on')
523         N = 50
524         t = np.linspace(0.01,np.pi-.01,N)
525         for i in range(N):
526             u0[0] = np.sin(t[i])
527             u0[2] = np.cos(t[i])
528             if u0[0] < 0:
529                 u0[0] = -.71+u0[0]
530             else:
531                 u0[0] = .71+u0[0]
532             print 'i=%d'%i, u0
533             X = solve(C,u0,s0,s1,ds,solver)
534             r = X[:,0]
535             theta = X[:,2]
536             #phi = X[:,4]
537             #x = r*np.sin(theta)*np.cos(phi)
538             #y = r*np.sin(theta)*np.sin(phi)
539             #z = r*np.cos(theta)
540             #ax.plot(x,y,z,'b')
541             t = X[:,4]
542             x = r*np.sin(theta)
543             y = r*np.cos(theta)
544             z = t
545             ax.plot(x,y,z,'b')
546
547     ax.set_xlabel('x')
548     ax.set_ylabel('y')
549     ax.set_zlabel('z')
550     plt.show()
551
552     return X,plt
553
554 def display_multiple_geodesics(u0,s0,s1,ds,surface,with_object=True):
555     import matplotlib.pyplot as plt
556     from mpl_toolkits.mplot3d import Axes3D
557     def solve_multiple(C,u0,s0,s1,ds,s):
558         from sympy.abc import u,v
559         print 'Running solver...'
560         return sc.odeint(f,u0,s,args=(C,u,v))
561
562     def display_multiple_catenoid(u0,s0,s1,ds,C,s):
563         X = solve_multiple(C,u0,s0,s1,ds,s)
564         plt.hold('on')
565         # plot the parametrized data on to the catenoid
566         u,v = X[:,0], X[:,2]
567         x = np.cos(u) - v*np.sin(u)
568         y = np.sin(u) + v*np.cos(u)
569         z = v
570         ax.plot(x,y,z,'--r')
571         return plt
572
573     def display_multiple_egg_carton(u0,s0,s1,ds,C,s):
574         X = solve_multiple(C,u0,s0,s1,ds,s)
575         plt.hold('on')
576         # plot the parametrized data on to the egg carton
577         u,v = X[:,0], X[:,2]
578         x = u
579         y = v

```

```

580     z = np.sin(u)*np.cos(v)
581     ax.plot(x,y,z,'--r')
582     return plt
583
584 def display_multiple_sphere(u0,s0,s1,ds,C,s):
585     X = solve_multiple(C,u0,s0,s1,ds,s)
586     plt.hold('on')
587     # plot the parametrized data on to the sphere
588     u,v = X[:,0], X[:,2]
589     x = np.cos(u)*np.cos(v)
590     y = np.sin(u)*np.cos(v)
591     z = np.sin(v)
592     ax.plot(x,y,z,'--r')
593     return plt
594
595 def display_multiple_torus(u0,s0,s1,ds,C,s):
596     X = solve_multiple(C,u0,s0,s1,ds,s)
597     plt.hold('on')
598     # plot the parametrized data on to the sphere
599     u,v = X[:,0], X[:,2]
600     x = (2 + 1*np.cos(v))*np.cos(u)
601     y = (2 + 1*np.cos(v))*np.sin(u)
602     z = np.sin(v)
603     ax.plot(x,y,z,'--r')
604     return plt
605
606 u0_range = np.arange(s0,s1+ds,ds)
607 N = u0_range.shape[0]
608
609 fig = plt.figure()
610 if surface == 'catenoid':
611     if with_object:
612         u,v = plt.meshgrid(np.linspace(-np.pi,np.pi,150),np.linspace(-np.pi,np.pi,150))
613         x = np.cos(u) - v*np.sin(u)
614         y = np.sin(u) + v*np.cos(u)
615         z = v
616     C = catenoid()
617 elif surface == 'egg_carton':
618     if with_object:
619         u,v = plt.meshgrid(np.linspace(-4,4,250),np.linspace(-4,4,250))
620         x = u
621         y = v
622         z = np.sin(u)*np.cos(v)
623     C = egg_carton()
624 elif surface == 'sphere':
625     if with_object:
626         u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
627         x = np.cos(u)*np.cos(v)
628         y = np.sin(u)*np.cos(v)
629         z = np.sin(v)
630     C = sphere()
631 elif surface == 'torus':
632     if with_object:
633         u,v = plt.meshgrid(np.linspace(0,2*np.pi,150),np.linspace(0,2*np.pi,150))
634         x = (2 + 1*np.cos(v))*np.cos(u)
635         y = (2 + 1*np.cos(v))*np.sin(u)
636         z = np.sin(v)
637     C = torus()
638 ax = fig.add_subplot(111, projection='3d')
639 ax.view_init(azim=65, elev=67)
640 if with_object:
641     theCM = 'Pastell1'
642     ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
643 plt.hold('on')
644
645 if surface == 'catenoid':
646     for u_val in u0_range:
647         u0[3] = u_val

```

```

648         plt = display_multiple_catenoid(u0,s0,s1,ds,C,u0_range)
649     elif surface == 'egg_carton':
650         for u_val in u0_range:
651             u0[0] = u_val
652             plt = display_multiple_egg_carton(u0,s0,s1,ds,C,u0_range)
653     elif surface == 'sphere':
654         for u_val in u0_range:
655             u0[0] = u_val
656             plt = display_multiple_sphere(u0,s0,s1,ds,C,u0_range)
657     elif surface == 'torus':
658         for u_val in u0_range:
659             u0[0] = u_val # alternate v0 values
660             plt = display_multiple_torus(u0,s0,s1,ds,C,u0_range)
661
662
663     from math import pi
664     s0_ = s0#/pi
665     s1_ = s1/pi
666     fig.suptitle("$s$ in [%1.f,%1.f]pi$ , $u$' = %1.f$ , $v$' = %1.f$ , $v$' = %1.f$"%(s0_,s1_,u0[1],u0[2],u0
667         [3]))
668     plt.show()
669
670 if __name__ == '__main__':
671     import sys
672     from math import pi, sqrt
673     if len(sys.argv) > 1:
674         u0 = eval(sys.argv[1]) # evaluate the input for a list [u(0), u'(0), v(0), v'(0)]
675         if type(u0) is not list:
676             raise TypeError("The first argument must be a list : [u(0),v(0),u'(0),v'(0)]")
677         s0 = float(eval(sys.argv[2])) # evaluate math expressions such as pi, '*', sqrt
678         s1 = float(eval(sys.argv[3]))
679         ds = float(eval(sys.argv[4]))
680         display = sys.argv[5]
681         if display == 'catenoid':
682             display_catenoid(u0,s0,s1,ds)
683         if display == 'sphere':
684             display_sphere(u0,s0,s1,ds)
685         if display == 'torus':
686             display_torus(u0,s0,s1,ds)
687         if display == 'egg_carton':
688             display_egg_carton(u0,s0,s1,ds)
689     else:
690         """
691         u0 = [0,-.5,.5,0] # u(0), u'(0), v(0), v'(0)
692         s0 = -pi/2
693         s1 = 3*pi
694         ds = 0.1
695         display_catenoid(u0,s0,s1,ds,show=True)
696         #display_multiple_geodesics(u0,s0,s1,ds,'catenoid',with_object=False)
697
698         u0 = [0.75,0.1,.75,0.1]
699         s0 = 0
700         s1 = 18*pi
701         ds = 2
702         #display_sphere(u0,s0,s1,ds,show=True)
703         display_multiple_geodesics(u0,s0,s1,ds,'sphere',with_object=False)
704
705         u0 = [0,.2,0,.2] # u(s0), u'(s0), v(s0), v'(s0)
706         s0 = 0
707         s1 = 25*pi
708         ds = .1
709         a = 1
710         c = -2
711         display_torus(u0,s0,s1,ds,a=c,show=True)
712         #display_multiple_geodesics(u0,s0,s1,ds,'torus',with_object=False)
713
714         u0 = [0,.5,0.5,sqrt(3)/2] # u(0), u'(0), v(0), v'(0)
715         s0 = -pi

```

```

715     s1 = pi
716     ds = 0.05
717     display_egg_carton(u0,s0,s1,ds,show=True)
718     #display_multiple_geodesics(u0,s0,s1,ds,'egg_carton')
719
720     u0 = [1.5,.1,-1,0] # u(0), u'(0), v(0), v'(0)
721     s0 = 0
722     s1 = 80
723     ds = 0.1
724     display_mobius_strip(u0,s0,s1,ds,show=True,solver=None)
725
726
727     u0 = [0,0,0,.1] # u(s0), u'(s0), v(s0), v'(s0)
728     s0 = 0
729     s1 = 10*pi
730     ds = 0.5
731     display_toroid(u0,s0,s1,ds,u_val=1,show=True)
732
733     # check if two points are connected on great circle
734     p1 = (1,1) # taken from a sphere simulation for u' = .2, v' = 0
735     p2 = (0.16242917238268037, 0.80611950949349132) # entry 200 in X
736     s0 = 0
737     s1 = 18*pi
738     ds = 0.05
739     C = sphere()
740     two_points(p1,p2,s0,s1,ds,C,tol=1e-6,surface='sphere')
741     """
742     u0 = [.7,.1,.1,.1,0,.1] # u(s0), u'(s0), v(s0), v'(s0), w(s0), w'(s0)
743     s0 = 0
744     s1 = 18
745     ds = 0.01
746     # A singularity near origo : a = 0, G = 1, M = 0.5
747     a = 0
748     G = 1
749     M = 0.35
750     display_3D_Kerr(u0,s0,s1,ds,show=True,solver=None,args = (a,G,M))

```

C.6 Hyperstreamlines

hyperstreamlines.py

```

1 from __future__ import division
2 import scipy.integrate as sc
3 import numpy as np
4 import numpy.linalg as nplin
5
6 def find_eigen(T):
7     """
8     Returns the eigenvalues and eigenvectors for a second order tensor T.
9     """
10    dim = T.shape[0]
11    eig_data = nplin.eig(T)
12    eig_values = eig_data[0]
13    v1 = eig_data[1][:,0]
14    v2 = eig_data[1][:,1]
15    if dim == 3:
16        v3 = eig_data[1][:,2]
17        return eig_values,v1,v2,v3
18    else:
19        return eig_values,v1,v2
20
21 from scipy.interpolate import griddata
22 def integrate(grid_points,U,p0,s,direction='major',solver=None):
23     dim = 2
24     if len(U) == 12:
25         dim = 3
26         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

```

```

27
28     if direction == 'major':
29         U__x = U_x.flatten(); U__y = U_y.flatten(); U__z = U_z.flatten();
30     else:
31         U__x = U_x_.flatten(); U__y = U_y_.flatten(); U__z = U_z_.flatten();
32     points = zip(grid_points[0].flatten(),grid_points[1].flatten(),grid_points[2].flatten())
33     def f_3D(x,t):
34         return [griddata(points,U__x,x)[0], griddata(points,U__y,x)[0], griddata(points,U__z,x)[0]]
35     f = f_3D
36 else:
37     U_x, U_y, U_x_, U_y_, l_minor, l_major = U
38     if direction == 'major':
39         U__x = U_x.flatten(); U__y = U_y.flatten();
40     else:
41         U__x = U_x_.flatten(); U__y = U_y_.flatten();
42     points = zip(grid_points[0].flatten(),grid_points[1].flatten())
43     def f_2D(x,t):
44         return [griddata(points,U__x,x)[0], griddata(points,U__y,x)[0]]
45     f = f_2D
46
47 if solver == None: # use lsoda from scipy.integrate.odeint
48     print 'Running solver ...'
49     p = sc.odeint(f,p0,s)
50 else: # use any other solver from scipy.integrate.ode
51     # vode,zvode,lsoda,dopri5,dop853
52     r = sc.ode(lambda t,x: f(x,t)).set_integrator(solver)
53     r.set_initial_value(p0)
54     y = []
55     t_end = s[-1]
56     dt = s[1]-s[0]
57     print 'Running solver ...'
58     while r.successful() and r.t <= t_end:
59         r.integrate(r.t + dt)
60         y.append(r.y)
61         p = np.array(y)
62 if direction == 'major':
63     if dim == 2:
64         p_ = _expand_hyperstreamline(p,U_x_,U_y_,l_minor,points)
65     else:
66         p_ = _expand_hyperstreamline3D(p,U_x_,U_y_,U_z_,U_x__,U_y__,U_z__,l_minor,l_medium,points)
67 else:
68     if dim == 2:
69         p_ = _expand_hyperstreamline(p,U_x,U_y,l_major,points)
70     else:
71         p_ = _expand_hyperstreamline3D(p,U_x,U_y,U_z,U_x__,U_y__,U_z__,l_major,l_medium,points)
72 return p,p_
73
74 def _expand_hyperstreamline(p,Ux,Uy,l,points):
75     p_ = np.zeros_like(p)
76     def f(x):
77         return [griddata(points,Ux,x)[0], griddata(points,Uy,x)[0]]
78     i = 0
79     for p_val in p:
80         print p_val
81         p_[i] = f(p_val)*l[p_val]
82         i = i + 1
83     return p_
84
85
86 def _expand_hyperstreamline3D(p,Ux,Uy,Uz,Ux_,Uy_,Uz_,l,l_,points):
87     p_ = np.zeros_like(p)
88     def f1(x):
89         return [griddata(points,Ux,x)[0], griddata(points,Uy,x)[0],griddata(points,Uz,x)[0]]
90     def f2(x):
91         return [griddata(points,Ux_,x)[0], griddata(points,Uy_,x)[0],griddata(points,Uz_,x)[0]]
92     return p_
93
94 def extract_eigen(eigen_field):

```



```

95     """
96     Performs a sorting of minor, major, and (if 3D) medium eigenvectors and eigenvalues.
97     The 'eigen_field' is assumed to be of the form
98
99         [[ l1, l2],
100          [v1x, v1y],
101          [v2x, v2y]]
102     and for 3D
103         [[ l1, l2, l3],
104          [v1x, v1y, v1z],
105          [v2x, v2y, v2z],
106          [v3x, v3y, v3z]]
107     Finally, the corresponding eigenvalues are returned as well.
108     """
109     return _sort_eig(eigen_field)
110
111 def _sort_eig(U):
112     dim = U.shape[1]
113     Nx = U.shape[2]
114     Ny = U.shape[3]
115     if dim == 3:
116         Nz = U.shape[4]
117         return _sort_eig_3D(U, Nx, Ny, Nz)
118
119     U_x = np.zeros([Nx, Ny]); U_y = np.zeros_like(U_x); # major eigenvectors
120     U_x_ = np.zeros_like(U_x); U_y_ = np.zeros_like(U_x); # minor eigenvectors
121     l_major = np.zeros_like(U_x); l_minor = np.zeros_like(U_x)
122     print 'Sorting eigenvalues and eigenvectors ...'
123     for i in range(Nx):
124         for j in range(Ny):
125             if U[0,0,i,j] <= U[0,1,i,j]: # if lambda_1 < lambda_2
126                 l_minor[i,j] = U[0,0,i,j]
127                 l_major[i,j] = U[0,1,i,j]
128                 U_x[i,j] = U[2,0,i,j]
129                 U_y[i,j] = U[2,1,i,j]
130                 U_x_[i,j] = U[1,0,i,j]
131                 U_y_[i,j] = U[1,1,i,j]
132             else:
133                 l_major[i,j] = U[0,1,i,j]
134                 l_minor[i,j] = U[0,0,i,j]
135                 U_x[i,j] = U[1,0,i,j]
136                 U_y[i,j] = U[1,1,i,j]
137                 U_x_[i,j] = U[2,0,i,j]
138                 U_y_[i,j] = U[2,1,i,j]
139     return U_x_, U_y_, U_x, U_y, l_minor, l_major
140
141 def _sort_eig_3D(U, Nx, Ny, Nz):
142     U_x = np.zeros([Nx, Ny, Nz]); U_y = np.zeros_like(U_x); U_z = np.zeros_like(U_x); # major
143     U_x_ = np.zeros_like(U_x); U_y_ = np.zeros_like(U_x); U_z_ = np.zeros_like(U_x); # minor
144     U_x__ = np.zeros_like(U_x); U_y__ = np.zeros_like(U_x); U_z__ = np.zeros_like(U_x); # medium
145     l_major = np.zeros_like(U_x); l_minor = np.zeros_like(U_x); l_medium = np.zeros_like(U_x);
146     print 'Sorting eigenvalues and eigenvectors ...'
147     for i in range(Nx):
148         for j in range(Ny):
149             for k in range(Nz):
150                 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]):
151                     l_major[i,j,k] = U[0,0,i,j,k]
152                     if U[0,1,i,j,k] >= U[0,2,i,j,k]:
153                         l_minor[i,j,k] = U[0,2,i,j,k]
154                         l_medium[i,j,k] = U[0,1,i,j,k]
155                         U_x_[i,j,k] = U[2,0,i,j,k]
156                         U_y_[i,j,k] = U[2,1,i,j,k]
157                         U_z_[i,j,k] = U[2,2,i,j,k]
158                         U_x__[i,j,k] = U[3,0,i,j,k]
159                         U_y__[i,j,k] = U[3,1,i,j,k]
160                         U_z__[i,j,k] = U[3,2,i,j,k]
161                     else:
162                         l_minor[i,j,k] = U[0,1,i,j,k]

```

```

163         l_medium[i,j,k] = U[0,2,i,j,k]
164         U_x_[i,j,k] = U[3,0,i,j,k]
165         U_y_[i,j,k] = U[3,1,i,j,k]
166         U_z_[i,j,k] = U[3,2,i,j,k]
167         U_x__[i,j,k] = U[2,0,i,j,k]
168         U_y__[i,j,k] = U[2,1,i,j,k]
169         U_z__[i,j,k] = U[2,2,i,j,k]
170         U_x[i,j,k] = U[1,0,i,j,k]
171         U_y[i,j,k] = U[1,1,i,j,k]
172         U_z[i,j,k] = U[1,2,i,j,k]
173     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]):
174         l_major[i,j,k] = U[0,1,i,j,k]
175         if U[0,0,i,j,k] >= U[0,2,i,j,k]:
176             l_minor[i,j,k] = U[0,2,i,j,k]
177             l_medium[i,j,k] = U[0,0,i,j,k]
178             U_x_[i,j,k] = U[3,0,i,j,k]
179             U_y_[i,j,k] = U[3,1,i,j,k]
180             U_z_[i,j,k] = U[3,2,i,j,k]
181             U_x__[i,j,k] = U[1,0,i,j,k]
182             U_y__[i,j,k] = U[1,1,i,j,k]
183             U_z__[i,j,k] = U[1,2,i,j,k]
184         else:
185             l_minor[i,j,k] = U[0,0,i,j,k]
186             l_medium[i,j,k] = U[0,2,i,j,k]
187             U_x_[i,j,k] = U[1,0,i,j,k]
188             U_y_[i,j,k] = U[1,1,i,j,k]
189             U_z_[i,j,k] = U[1,2,i,j,k]
190             U_x__[i,j,k] = U[3,0,i,j,k]
191             U_y__[i,j,k] = U[3,1,i,j,k]
192             U_z__[i,j,k] = U[3,2,i,j,k]
193         U_x[i,j,k] = U[2,0,i,j,k]
194         U_y[i,j,k] = U[2,1,i,j,k]
195         U_z[i,j,k] = U[2,2,i,j,k]
196     else:
197         l_major[i,j,k] = U[0,2,i,j,k]
198         if U[0,0,i,j,k] >= U[0,1,i,j,k]:
199             l_minor[i,j,k] = U[0,1,i,j,k]
200             l_medium[i,j,k] = U[0,0,i,j,k]
201             U_x_[i,j,k] = U[2,0,i,j,k]
202             U_y_[i,j,k] = U[2,1,i,j,k]
203             U_z_[i,j,k] = U[2,2,i,j,k]
204             U_x__[i,j,k] = U[1,0,i,j,k]
205             U_y__[i,j,k] = U[1,1,i,j,k]
206             U_z__[i,j,k] = U[1,2,i,j,k]
207         else:
208             l_minor[i,j,k] = U[0,0,i,j,k]
209             l_medium[i,j,k] = U[0,1,i,j,k]
210             U_x_[i,j,k] = U[1,0,i,j,k]
211             U_y_[i,j,k] = U[1,1,i,j,k]
212             U_z_[i,j,k] = U[1,2,i,j,k]
213             U_x__[i,j,k] = U[2,0,i,j,k]
214             U_y__[i,j,k] = U[2,1,i,j,k]
215             U_z__[i,j,k] = U[2,2,i,j,k]
216         U_x[i,j,k] = U[3,0,i,j,k]
217         U_y[i,j,k] = U[3,1,i,j,k]
218         U_z[i,j,k] = U[3,2,i,j,k]
219     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
220
221 def _run_example_flat_sphere(xstart,xend,N,direction='major',solver=None):
222     """
223     A test example, using the metric of a flat sphere, to calculate hyperstreamlines
224     for a 2D grid.
225     """
226     x0,y0 = xstart
227     xN,yN = xend
228     Nx,Ny = N
229     x,y = np.mgrid[x0:xN:Nx*1j,y0:yN:Ny*1j]
230     # Initialize the metric for the flat sphere

```

```

231 g = np.array([[1,0],[0,1]],dtype=np.float32)
232 T = np.zeros([2,2,Nx,Ny],dtype=np.float32) # The tensor field
233 eig_field = np.zeros([3,2,Nx,Ny],dtype=np.float32) # The "eigen" field
234
235 print "Determining eigenvectors for the flat metric of a sphere over the mesh..."
236 for i in range(Nx):
237     for j in range(Ny):
238         g[1,1]= np.sin(y[i,j])**2
239         T[:, :, i,j] = g[:, :]
240         eig_field[:, :, i,j] = find_eigen(T[:, :, i,j])
241
242 INITIAL_POINT = (1.,1.)
243 t0 = 0
244 t1 = 2*np.pi
245 dt = 0.01
246 t = np.arange(t0,t1+dt,dt)
247 U = extract_eigen(eig_field)
248 p,p_ = integrate([x,y],U,INITIAL_POINT,t,direction=direction,solver=solver)
249 return p,p_
250
251 def _run_example_3D(xstart,xend,N,direction='major',solver=None):
252     """
253     A 3D test example
254     """
255     x0,y0,z0 = xstart
256     xN,yN,zN = xend
257     Nx,Ny,Nz = N
258     x,y,z = np.mgrid[x0:xN:Nx*1j,y0:yN:Ny*1j,z0:zN:Nz*1j]
259     # Initialize the metric for the flat sphere
260     g = np.array([[1,0,0],[0,.5,0],[0,0,1]],dtype=np.float32)
261     T = np.zeros([3,3,Nx,Ny,Nz],dtype=np.float32) # The tensor field
262     eig_field = np.zeros([4,3,Nx,Ny,Nz],dtype=np.float32) # The "eigen" field
263
264     print "Determining eigenvectors for the flat metric of a sphere over the mesh..."
265     for i in range(Nx):
266         for j in range(Ny):
267             for k in range(Nz):
268                 g[2,2]= np.sin(y[i,j,k])**2 + np.cos(x[i,j,k])**2
269                 T[:, :, i,j,k] = g[:, :]
270                 eig_field[:, :, i,j,k] = find_eigen(T[:, :, i,j,k])
271
272     INITIAL_POINT = (1.,1.,1.)
273     t0 = 0
274     t1 = 2*np.pi
275     dt = 0.01
276     t = np.arange(t0,t1+dt,dt)
277     U = extract_eigen(eig_field)
278     p,p_ = integrate([x,y,z],U,INITIAL_POINT,t,direction=direction,solver=solver)
279     return p,p_
280
281 if __name__ == "__main__":
282     import sys
283     x0 = 0; y0 = 0; z0 = 0
284     xN = np.pi/2; yN = np.pi; zN = 1
285     Nx = 22; Ny = 22; Nz = 22
286     N = (Nx,Ny,Nz)#N = (Nx,Ny)
287     xstart = (x0,y0,z0); xend = (xN,yN,zN)
288     #xstart = (x0,y0); xend = (xN,yN)
289     solver = None # solvers: lsoda (default), vode,zvode,lsoda,dopri5,dopri853
290
291     if len(sys.argv) > 1:
292         if sys.argv[1] == "major":
293             p,p_ = _run_example_flat_sphere(xstart,xend,N,'major',solver=solver)
294         else:
295             p,p_ = _run_example_flat_sphere(xstart,xend,N,'minor',solver=solver)
296     else:
297         #p,p_ = _run_example_flat_sphere(xstart,xend,N,'major',solver=solver)
298         p,p_ = _run_example_3D(xstart,xend,N,'major',solver=solver)

```

```
299
300     import matplotlib.pyplot as plt
301     from mpl_toolkits.mplot3d import Axes3D
302     fig = plt.figure()
303     ax = fig.add_subplot(111, projection='3d')
304     plt.plot(p[:,0],p[:,1],p[:,2])
305     plt.show()
```