1 Implementation

Currently there are quite limited options when it comes to software which can visualize second order (or higher order) tensor fields. This problem is further exacerbated when it comes to free software. MayaVi[1] offers the possibilty to display hyperstreamlines, and VisIt[2] allows visualization by tensor ellipsoids. Both MayaVi and VisIt have a Python module, and they also permit users to have the (optional) interface to interact with data. However, both of these software offer visualization methods which require a symmetric tensor field. This in turn limits data a user can visualize; asymmetric tensor fields can have complex eigenvalues, which can not be handeled by either methods [**?**].

As such we have created our own software, where we include features such as hyperstreamlines and hybrid geodesic-streamline visualization.

1.1 Geodesic Differential Equations

The geodesic differential equations in two-dimensions[3] are given as

$$u'' + \left\{ \begin{matrix} 0 \\ 0\;0 \end{matrix} \right\} (u')^2 + 2\left\{ \begin{matrix} 0 \\ 0\;1 \end{matrix} \right\} u'v' + \left\{ \begin{matrix} 0 \\ 1\;1 \end{matrix} \right\} (v')^2 = 0,$$
$$u'' + \left\{ \begin{matrix} 1 \\ 0\;0 \end{matrix} \right\} (u')^2 + 2\left\{ \begin{matrix} 1 \\ 0\;1 \end{matrix} \right\} u'v' + \left\{ \begin{matrix} 1 \\ 1\;1 \end{matrix} \right\} (v')^2 = 0.$$

where $\left\{ \begin{smallmatrix} m \\ i\;j \end{smallmatrix} \right\} = \left\{ \begin{smallmatrix} m \\ i\;j \end{smallmatrix} \right\} (u(s), v(s))$ is the Christoffel symbol of second kind evaluated at (u, v).

The geodesic solution $u = u(s), v = v(s)$ is a curve defined over the interval $s \in [s_0, s_1]$. We can rewrite these equations as a system of first order differential equations by defining p and q, such that $p = u'$, and $q = v'$

$$p' + \left\{ \begin{matrix} 0 \\ 0\;0 \end{matrix} \right\} p^2 + 2\left\{ \begin{matrix} 0 \\ 0\;1 \end{matrix} \right\} pq + \left\{ \begin{matrix} 0 \\ 1\;1 \end{matrix} \right\} q^2 = 0,$$
$$q' + \left\{ \begin{matrix} 1 \\ 0\;0 \end{matrix} \right\} p^2 + 2\left\{ \begin{matrix} 1 \\ 0\;1 \end{matrix} \right\} pq + \left\{ \begin{matrix} 1 \\ 1\;1 \end{matrix} \right\} q^2 = 0.$$

Given the initial conditions $u(s_0) = u_0$, $v(s_0) = v_0$, $u'(s_0) = p_0$, $v'(s_0) = q_0$, we can solve the boundary value problem numerically. A solver from *scipy.integrate*[4] is used to solve the following system of second order differential equations. Here is a code snippet for the numerical solver.

```
def f(y,s,C,u,v):
        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
```

[1] `http://docs.enthought.com/mayavi/mayavi/overview.html`
[2] `http://www.visitusers.org/index.php?title=Main_Page`
[3] We can readily extend this problem to N-dimensions.
[4] http://docs.scipy.org/doc/scipy/reference/integrate.html

```

        C = C.subs({u:y0,v:y2}) # Evaluate C for u,v = (u0,v0)

        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):
        s = np.arange(s0,s1+ds,ds)
        # The Christoffel symbol of 2nd kind, C, is a symbolic function of (u,v)
        from sympy.abc import u,v
        return sc.odeint(f,u0,s,args=(C,u,v)) # integration method : LSODA
```

To determine the Christoffel symbol of second kind, we use the symbolic Python package *SymPy*[1]. The following script demonstrates how a user can create Christoffel symbols of both first and second kind.

```
from sympy import symbols, sin
from sympy.diffgeom import Manifold, Patch, CoordSystem, TensorProduct
from sympy.diffgeom import metric_to_Christoffel_1st, metric_to_Christoffel_2nd

dim = 2
m = Manifold("M",dim)
patch = Patch("P",m)

flat_sphere = CoordSystem("flat_sphere", patch, ["theta", "phi"])
theta, phi = flat_sphere.coord_functions()
dtheta,dphi = flat_sphere.base_oneforms()

r = sympy.symbols('r')
metric_diff_form = r**2*TensorProduct(dtheta, dtheta) + r**2*sin(theta)**2*TensorProduct(dphi, dphi)

C1 = metric_to_Christoffel_1st(metric_diff_form)
C2 = metric_to_Christoffel_2nd(metric_diff_form)
```

We have implemented a tensor module which allows the user to state a metric, g_{ij}, which can then be used to generate the corresponding Christoffel symbols. The module permits the user to also find the Riemann-Christoffel tensor, the Ricci tensor, and the scalar-curvature. The module is found in the appendix **??**.

1.2 The 3D Kerr Metric

Several metric examples are stored in file find_metric.py **??**. This module contains other convenient functions as well; for instance the ability to specify a curve element to generate the corresponding metric tensor.

Consider the Kerr solution[2] expressed in terms of polar coordinates r, θ, ϕ, such that $x = r\sin(\theta)\cos(\phi)$, $y = r\sin(\theta)\sin(\phi)$, $z = r\cos(\theta)$. Then the Kerr metric is given as

$$\begin{aligned} ds^2 = &-\left(1 - \frac{2GMr}{r^2 + a^2\cos^2(\theta)}\right) dt^2 + \left(\frac{r^2 + a^2\cos^2(\theta)}{r^2 - 2GMr + a^2}\right) dr^2 + \left(r^2 + a^2\cos(\theta)\right) d\theta^2 \\ &+ \left(r^2 + a^2 + \frac{2GMra^2}{r^2 + a^2\cos^2(\theta)}\right)\sin^2(\theta) d\phi^2 - \left(\frac{4GMra\sin^2(\theta)}{r^2 + a^2\cos^2(\theta)}\right) d\phi\, dt \end{aligned}$$

[1]http://docs.sympy.org/dev/modules/diffgeom.html

[2]The Kerr metric is useful for many calculations regarding objects near to rotating planets and ordinary stars (it even applies to objects with strong fields, such as black holes or neutron stars)[**?**].

where $a \equiv S/M$ is the object's angular momentum[1] per unit mass, and G is the gravitational constant. This is an exact solution for the empty-space Einstein equation. Therefore, this solution is very important in astrophysics. The Kerr solution describes the unique geometry of spacetime outside of of any[2] axially symmetric object (which includes blackholes!).

Here is an implementation for defining a Kerr metric by it's curve element.

```
from sympy import symbols, simplify, cos, sin
from sympy.abc import G,M,l,u,v,w,c
import find_metric as fm
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 = fm.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})

# perform any possible simplifications on the metric
g = simplify(g)
```

We can display geodesics for this tensor, if we consider the three-dimensional case, where we keep the time frame constant, or one of the space coordinates.

```
def 3D_kerr_constant_space(a=0,G=1,M=0.5):
    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 = 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 3D_kerr_constant_time(a=0,G=1,M=0.5):
    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)))
    metric = g11*TensorProduct(du, du) + g22*TensorProduct(dv, dv) + g33*TensorProduct(dw, dw)
    C = Christoffel_2nd(metric=metric)
    return C
```

We have implemented a function which can handle 3D cases, just like we have demonstrated 2D above.

[1]If $a = 0$, the solution reduces to the Schwarzchild solution(this is also included in the *find_metric.py* module).

[2]With some limitations attached. See [**?**].

1.3 Hyperstreamlines

We have implemented a hyperstreamline module which can determine hyperstreamlines for a user provided seed point, and a predefined direction (either major or minor eigenvector). Here is an example, where we find the hyperstreamlines for a metric of a sphere on a 2D surface

```
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_
```