Visualization of Second Order Tensor Fields

by

Imran Ali

Thesis

for the degree of

Master of Science

(Master i Anvendt Matematikk og Mekanikk)

Faculty of Mathematics and Natural Sciences
University of Oslo

November 2015

Det matematisk- naturvitenskapelige fakultet
Universitetet i Oslo

Abstract

Geodesics and hyperstreamlines are used to visualize second order tensors. We further look at a new way of visualizing second order tensor fields. By using the direction of geodesic curves in stead of eigenvectors, we make a different approach to so called integration methods. We extend the concept to include tensors which are not necessarily the metric.

Keywords: Tensor Field Visualization, Hyperstreamlines, Geodesics

Acknowledgements

Thesis supervisors :

- Øyvind Andreassen (FFI,UNIK),
- Anders Helgeland (FFI,UNIK),
- Kent-Andre Mardal (UiO).

This thesis uses extensively the programming language Python. For the sake of reproducibility, here we list the versions of the packages and sub-packages that are used throughout the thesis.

- Python - v.2.7.9
  - NumPy - v.1.8.2
  - SciPy - v.0.16.0
  - Matplotlib - v.1.4.2
  - SymPy - v.0.7.7.dev
- IPython - v.2.3.0

For SymPy, instead of using a stable release[1], we were required to use the latest developer version due to bugs that were present in the current release[2].

This may seem like a strange thing to do, but I would like to acknowledge the following. A high-level language like Python has made this thesis a very fun and interesting task. Instead of bogging down into details, Python has given me the leverage required to focus on other aspects of the coding; where you are no longer required to exhibit strenuous efforts to locate bugs and errors. Instead it allows the coder to focus efforts in quickly writing down the mathematical or physical problem with relative ease, and visualising the results in a similar quick manner. The best part in all of this is that every package and sub-package used to achieve the results are free and available online. Therefore, I am grateful to every person who has contributed in making Python into the powerful mathematical tool that it has now become.

Last but not least, I am grateful to all my supervisors for their support and encouragement. Frankly, I am befuddled that they managed to put up with my constant queries. In this regard, to more than any, I am grateful to Professor Øyvind Andreassen.

[1] https://github.com/sympy/sympy/releases
[2] https://github.com/sympy/sympy/releases/tag/sympy-0.7.6.1

Contents

List of Figures

List of Tables

Part I
Theory

1 Introduction

Currently, data produced from simulations and experiments is ever increasing in size and complexity. As computers become faster and more (fast) memory becomes available, data generation increases in tact with the technological advances. Scientists often produce terabytes[1] of data from "simple" simulations. In order to process this data several approaches can be made. The data can be stored on a hard disk drive and thereupon accessed from any program. Users can either create their own software to process the data, or use prebuilt software intended for post processing. Depending on the complexity of the data, such software will have it's limitations. Software engineers and computer scientists often require specific features which suite their needs. A promising language like Python is becoming increasingly popular. It is an object-oriented language which offers an almost mathematical stylistic code. This makes code readability that much easier for someone with a natural science background. Even though Python is an interpreted language, it is still possible to match languages such as Fortran and C in computational speed. Cython[2] provides the possibility to combine the efficiency of C with the ease of readability of Python. This combination is quite potent, as the computer scientist can possess the speed and still maintain few lines of code with mathematical tones.

Our goal for this thesis will be to study and employ methods specifically for visualization of tensor fields. In general, this implies that given some data (regardless of it's dimension and order) how can we portray the data in such a way that the human mind can perceive it logically with as little effort as possible? Perception of complex structured data is the driving force behind this thesis. The challenge here lies in finding intuitive ways of visualizing data when the order or rank of the tensor increases. However, as the order of the tensor increases so does the difficulty in our ability to visualize and interpret the data. As such we have choosen to limit ourselves to second order tensors in $\mathbf{R}^2$ and $\mathbf{R}^3$. Instead we focus on applying visualization techniques on a variety of data structures (this is not a trivial task to accomplish!). In order to visualize the results, we have created our own software.

There are three common data structures : **scalar**, **vectorial**, and **tensorial**. Scalar data can often be visualized by employing color. For example, temperature and pressure can be visualized where the red color entails high values, and blue color low values. Every other value falls in between these colors; see for example Figure 1. As to the question : why exactly these colors? The apparent use of these colors stem from psycological reasonings. If someone desires to use an opposite color scheme, there are no restrictions to visualize high and low values with the aforementioned color gambit. As a matter of fact, it simply comes down to personal choices that an individual might find aesthetically pleasing. As such, technically speaking, there are no limitations, but some standards do exist.

Often the scalar values are mapped to a color model. Given some data, we can assign each value to a specific color through a mapping function between a color model and a refer-

[1]A house hold computer or desktop has a portion of this capacity as total storage.

[2]`http://cython.org/`

[1]Source : `http://earthobservatory.nasa.gov/IOTD/view.php?id=3505`

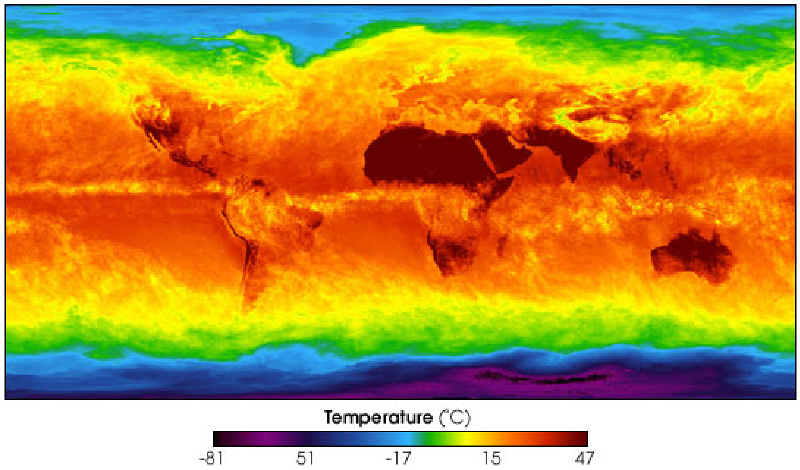


Figure 1: Picture taken from NASA Earth Observatorium.[1]

ence color space. Among some well known color maps are Jet, Parula[2], and Gray. Another color map is Viridis. Viridis has been developed for Python and has recently been added to Matplotlib as the default color map (intended for the release of version 2.0 [3]).

Some color models which are worth mentioning :

- Computer monitors employ the sRGB model. The three primary colors red, green, and blue are used to reproduce an array of colors. Unlike the standard RGB model, sRGB also defines a nonlinear transformation between the intensities of the three colors and the number stored.

- The CMY is a subtractive model, which uses the three primary colors cyan, magneta and yellow. This color model is often employed by printers.

- HSL and HSV are cylindrical coordinates representations of points in a RGB model. HSL stands for hue, saturation, and lightness. While HSV stand for hue, saturation, and value. Hue and saturation models are often used in image analysis. As such they are important models to be aware of.

Other techniques for 2D scalar data include *contour curves*, which are often used in cartography to display height differentials above for example sea level of similar values. For 3D scalar data similar techniques are used, with planes cutting through equal values, called *isosurfaces*.

For vectorial data in $\mathbf{R}^2$ and $\mathbf{R}^3$, an arrow entails to the length and direction of the data. Such data can be velocity and rotation for fluid flow, direction and strength of a magnetic field, or gradient of a scalar field (for a differentiable scalar function; e.g. temperature field). This is also referred to as glyph-based technique. Depiciting such data with glyphs can quickly lead to clutter or occlusion of the field view. As such, glyph-based techniques are best suited for small datasets. We can also use field lines or path lines, to visualize the flow of a vector

[2]Which is the current default color map in Matlab.

[3]`http://matplotlib.org/style_changes.html`

field. These flows often reveal the orientation of the field itself. Again, the amount of data limits the use of such techniques. Another issue lies in guessing proper seed points to extract important features. This is not an easy task if the properties of the field are unknown. One way to circumvent the problem of seeding and clutter is to use the LIC method[CL93]. LIC stands for line integral convolution. This technique involves convoluting the vector field onto some noisy data. This generates dense visualization of the field lines along the vectors.

Tensorial data can be of any order. In fact, scalar values are considered zeroth-order tensors, while vectors are considered to be first-order tensors (given that certain criterias are met, which we will discuss later in Chapter 2). In this thesis we will focus on second-order tensors in $\mathbf{R}^2$ and $\mathbf{R}^3$. Such data can include stress and strain tensors, diffusion tensor for magnetic resonance (DT-MRI) in medical imaging, metric tensors in differential geometry, Reynolds-stress tensor for modelling turbulence, and many other tensor fields. As such, we have a diverse amount of physical problems we can visualize, even though we are limiting ourselves to second rank tensors.

We provide a quick reference over some of these tensors in Chapter 3.

Before we start discussing and dwelling deeper in the field of visualization of tensor fields, a review is required of basic theory underlying tensors. In order to understand tensors and their visualization, the reader needs to be familiar with vector and matrix operations (see Appendices A and B). In Chapter 2, we will focus on the underlying theory for tensor fields. In Chapter 3 the current research in the field of visualization of tensor fields is explored. In Chapter 4, we divert our attention to the implementation of software for various visualization methods. The results are portrayed and discussed in Chapter 5. The final section, Chapter 6, contains various conclusions we draw based upon the findings from Chapter 3 and Chapter 5. All the relevant code listings are found in the Appendix C.

The source code has been made accessible on the following repository : `https://github.com/imranal/DiffTens`.

2 Tensor Fields

Our focus in this thesis is visualization of sceond order tensors in $\mathbf{R}^2$ and $\mathbf{R}^3$. However, the theory we present in the following subsections will not be curtailed to second order tensors, but will be applicable to any order tensor field. In order to achieve this, we require a general notation. In this chapter, we first introduce the definition of a tensor of type (M, N). Using this definition we show how tensors can be differentiated. We require this in order to introduce the Riemann Christoffel tensor; also known as Riemann curvature tensor, which quantifies the relative variation of initially parallel geodesics [Moo10]. Since such geodesics only variate relative to each other in a curved space, the Riemann tensor will be zero everywhere in flat space.

2.1 Notation

Here we provide a short introduction to Einstein notation, which we will employ throughout this thesis.

In an orthogonal coordinate system we can write a vector $\mathbf{A}$ in component form

$$\mathbf{A} = A_1\hat{e}_1 + A_2\hat{e}_2 + A_3\hat{e}_3,$$

where $\hat{e}_i,\ i = 1, 2, 3$ are the orthogonal unit vectors. The short hand notation we just employed for the orthogonal vectors uses a dummy subscript i. Using this subscript, we can achieve a shorter notation, also called the *index* notation :

$$A_i,\ i = 1, 2, 3.$$

Here A_i refers to all the components of the vector[1] $\mathbf{A}$.

To avoid confusion, a system of any order when raised to a power will be enclosed in paranthesis

$$(B^2)^3, ({y_1}^1)^2, ({T_{ij}}^{kl})^{1/2}.$$

Hence, the component B^2 is raised to the power of 3, the component ${y_1}^1$ is raised to the power of 2, and for the tensor ${T_{ij}}^{kl}$ we take the square root[2].

When a system has indices that occur unrepeated, it is implicitly understood that each of the subscripts and superscripts can take any of the integer values $1, \ldots, N$. For example the

[1] As a single subscript/superscript denotes the components of a vector, we can extend this to higher(or lower) order systems. No subscript or superscript denote scalars, which we refer to as zeroth-order systems. While two indices imply a second order system, which in the language of linear algebra is called a matrix (if the indices are not mixed). We will, however, refer to such systems as tensors. In fact, nomenclature tensor is the unified reference to all such systems, and those of higher order.

[2] Which is not as simple or straight forward as it sounds. In order for any mathematical operation to be performed on a tensor, the resulting tensor must obey the transformation law. We will discuss this later when we introduce the general definition of tensors.

Kronecker delta symbol δ_{ij}, defined by

$$\delta_{ij} = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j, \end{cases} \tag{1}$$

with $i, j = 1, 2, 3$, represents nine values. The indices i and j are called *free indices* and can take on any of the values specified by a given range.

The summation convention states that when an index which is repeated twice on the same side of an equation, it is understood to represent a summation of the repeated indices. Hence, a repeated index is called a *summation index*, while an unrepeated index is called a *free index*. To sum the diagonal entries of the Kroncker delta[1], we simply repeat the indices

$$\delta_{ii} = \delta_{11} + \delta_{22} + \delta_{33} = 3.$$

Note that we implicitely understand, when we write δ_{ii}, to mean that we are performing the summation

$$\sum_{i}^{3} \delta_{ii}.$$

Often when performing certain operations, the alternating tensor becomes a handy tool to possess. It is a short-hand notation, just like the Kronecker delta.

Definition. *The e-permutation sysmbol is defined as*

$$e_{ijk} = \begin{cases} 1 & \text{if } ijk \text{ is an even permutation,} \\ -1 & \text{if } ijk \text{ is an odd permutation,} \\ 0 & \text{when indices overlap.} \end{cases}$$

The definition is also applicable for larger sets as well. Another identity which is useful, is the e-δ identity. Given e_{ijk} the e-permutation symbol, and δ_{ij} the Kronecker delta, then for $i, j, k, m, n = 1, 2, 3$,

$$e_{ijk}e_{imn} = \delta_{jm}\delta_{kn} - \delta_{jn}\delta_{km},$$

where i is the summation index and j, k, m, n are the free indices. A mixed superscript and subscript identity, is defined as

$$e^{ijk}e_{imn} = \delta^{j}{}_{m}\delta^{k}{}_{n} - \delta^{j}{}_{n}\delta^{k}{}_{m}$$

Using the e-permutation symbol we can describe the determinant of a tensor A with the components a_{ij}, where $i, j = 1, \ldots, n$. Writing the determinant in index notation, it becomes

$$detA = \|A\| = e_{i_1 \ldots i_n} a_{1i_1} \ldots a_{ni_n}$$

For a 3×3 matrix,

$$\begin{aligned} detA &= e_{ijk}a_{1i}a_{1j}a_{1k} \\ &= e_{1jk}a_{11}a_{1j}a_{1k} + e_{2jk}a_{12}a_{1j}a_{1k} + e_{3jk}a_{13}a_{1j}a_{1k} \\ &= e_{123}a_{11}a_{12}a_{13} + e_{132}a_{11}a_{13}a_{12} + e_{213}a_{12}a_{11}a_{13} + e_{231}a_{12}a_{13}a_{11} + e_{312}a_{13}a_{11}a_{12} + e_{321}a_{13}a_{12}a_{11} \\ &= a_{11}a_{12}a_{13} - a_{11}a_{13}a_{12} - a_{12}a_{11}a_{13} + a_{12}a_{13}a_{11} + a_{13}a_{11}a_{12} - a_{13}a_{12}a_{11} \end{aligned}$$

This is a simple illustration for the need of using short hand notation.

[1]In linear algebra this is referred to as taking the *trace* of an $n \times n$ square matrix.

2.2 A General Definition of Tensor

We now introduce a definition of a tensor field, evaluated at a point $x^j = x^j(\bar{x}^j)$. The bar quantity are coordinates in different coordinate system. We assume that a mapping exists, such that $\bar{x}^j = \bar{x}^j(x^j)$. We define the transformation between the barred coordinate system and the barless coordinates as

$$\bar{x}^j = \bar{x}^j(x^j). \tag{2}$$

Definition. *A tensor of rank* $(M+N)$, $\mathcal{T}_{i_1 \dots i_M}{}^{j_1 \dots j_N}(x^j)$, *defined on a point* P *of a differential manifold* X_n *exists, if, under the coordinate transformation* (2), *the components transform according to the law*

$$\overline{\mathcal{T}}_{m_1 \dots m_M}{}^{n_1 \dots n_N} = \frac{\partial x^{i_1}}{\partial \bar{x}^{m_1}} \cdot \dots \cdot \frac{\partial x^{i_M}}{\partial \bar{x}^{m_M}} \cdot \frac{\partial \bar{x}^{n_1}}{\partial x^{j_1}} \cdot \dots \cdot \frac{\partial \bar{x}^{n_N}}{\partial x^{j_N}} \mathcal{T}_{i_1 \dots i_M}{}^{j_1 \dots j_N}. \tag{3}$$

The tensor is said to be covariant *of order M and* contravariant *of order N, if it obeys the* transformation law.

Example. Let $\phi = \phi(x^1, \dots, x^N)$ denote a tensor of type $(0,0)$. Then from Equation (3) it follows that

$$\bar{\phi}(\bar{x}^1, \dots, \bar{x}^N) = \phi(x^1, \dots, x^N).$$

This is simply a scalar. Hence scalars are zero order tensors. The gradient of the scalar is another tensorial quantity. By chain rule,

$$\frac{\partial \bar{\phi}}{\partial \bar{x}^j} = \frac{\partial x^i}{\partial \bar{x}^j} \frac{\partial \phi}{\partial x^i}$$

which we see is a covariant vector of type (1,0). □

In general, we can write such vectors by the transformation law.

$$\bar{A}_j = \frac{\partial x^i}{\partial \bar{x}^j} A_i.$$

Similarly, a contravariant vector is a type (0,1) tensor, and by the transformation law given as

$$\bar{A}^j = \frac{\partial \bar{x}^j}{\partial x^i} A^i.$$

The importance of tensors is made clear due to the fact that they are invariant under coordinate transformations. Hence, if all the components of the tensor vanish in one coordinate system, they will do so in any other system. Similarly, if the tensor is symmetric in one coordinate system, the property will perpetuate on to another coordinate system.

Proof. Given the tensor R_{ijk}, it is said to be symmetric in two of its indices if the components are unchanged when the indices are interchanged. For example, the third order tensor R_{ijk} is symmetric in the indices i and k, if

$$R_{ijk} = R_{kji} \tag{4}$$

for all values of i, j and k. By the transformation law (3),

$$\overline{R}_{lmn} = \frac{\partial x^i}{\partial \bar{x}^l}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^k}{\partial \bar{x}^n}R_{ijk}, \tag{5}$$

and

$$\overline{R}_{nml} = \frac{\partial x^k}{\partial \bar{x}^n}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^i}{\partial \bar{x}^l}R_{kji}.$$

By Equation (4) and (5) it follows that

$$\overline{R}_{nml} = \frac{\partial x^k}{\partial \bar{x}^n}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^i}{\partial \bar{x}^l}\left(\frac{\partial \bar{x}^l}{\partial x^i}\frac{\partial \bar{x}^m}{\partial x^j}\frac{\partial \bar{x}^n}{\partial x^k}\overline{R}_{lmn}\right) = \overline{R}_{lmn},$$

where we used the inverse relation of Equation (5). □

A tensor is skew-symmetric in two of its indices if the components are transformed to their negative values when the indices are interchanged. Using R_{ijk} again to illustrate this,

$$R_{ijk} = -R_{kji}$$

for all values of i, j and k. Using the proof above, we can readily show for a skew-symmetric tensor the property of invariance from one coordinate system to another.

2.3 Tensor Operations

In order for an operation to make sense when performed on a tensor, the resulting tensor must satisfy the transformation law (3).

Example. We can add or subtract similar components for different tensors. The following operation is permitted only when the indices match

$$C^i{}_{jk} = A^i{}_{jk} + B^i{}_{jk}. \tag{6}$$

This is invalid

$$A^i + B_j. \tag{7}$$

For (7), the system B is a covariant of order 1, while system A is contravariant of order 1.

We can show that the operation performed in (6) is correct by employing the transformation law. It is a good way to show that a certain tensor operation is valid. Let A and B be expressed by the transformation law (3)

$$\bar{A}^l{}_{mn} = \frac{\partial \bar{x}^l}{\partial x^i}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^k}{\partial \bar{x}^n}A^i{}_{jk}, \tag{8}$$

and,

$$\overline{B}^{l}{}_{mn} = \frac{\partial \bar{x}^l}{\partial x^i}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^k}{\partial \bar{x}^n}B^i{}_{jk}. \tag{9}$$

Then,

$$\overline{C}^{l}{}_{mn} = A^l{}_{mn} + B^l{}_{mn} = \frac{\partial \bar{x}^l}{\partial x^i}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^k}{\partial \bar{x}^n}\left(A^i{}_{jk} + B^i{}_{jk}\right) = \frac{\partial \bar{x}^l}{\partial x^i}\frac{\partial x^j}{\partial \bar{x}^m}\frac{\partial x^k}{\partial \bar{x}^n}C^i{}_{jk}.$$

Which clearly satisfies the transformation law.

For the case in (7), we can use same procedure to demonstrate that the quantity is not tensorial. In general, we can say that only tensors of same type (r, s) can be added together.

Multiplication (outer product) on the other hand does not require tensors have the exact same type. For example the outer product of the systems $A_m{}^i$ and B^{jkl} result in the new system $C_m{}^{ijkl}$,

$$C_m{}^{ijkl} = A_m{}^i B^{jkl}.$$

The newly constructed system from the outer product is a fifth order system, consisting of all the possible products from the components of $A_m{}^i$ with B^{jkl}.

The operation of contraction occurs when the lower and upper index are set equal to each other and the summation convention is thereby applied. Using the system $C_m{}^{ijkl}$ from the previous example, we can perform contraction on the lower index n with the upper index i as following

$$C_m{}^{mjkl} = C_1{}^{1jkl} + \cdots + C_N{}^{Njkl} = D^{jkl}$$

where we have summed the same indices through the summation convention. As a result of contracting the system C, the resulting system is now a third order system. In fact, performing contraction on a system always lowers the order of the system by two.

An *inner product* between two tensors is performed as following. Firstly, take the outer product of the tensor, thereafter perform a contraction on two of the indices. As contraction requires both super and sub-script, the tensors involved in such an operation must be at least of rank 1. Given two vectors A^i and B_j, their inner product is found by first performing an outer product

$$C^i{}_j = A^i B_j.$$

Thereupon, we perform a contraction by setting the indices equal to each other, and sum all the terms. Using the transformation law for the contravariant tensor A^i and covariant tensor B_j, they take the form

$$\bar{A}^i = \frac{\partial \bar{x}^j}{\partial x^m}A^m,$$
$$\overline{B}_j = \frac{\partial x^n}{\partial \bar{x}^j}B_n.$$

There upon we perform the product

$$\bar{A}^i\overline{B}_j = \left(\frac{\partial \bar{x}^j}{\partial x^m}A^m\right)\left(\frac{\partial x^n}{\partial \bar{x}^j}B_n\right),$$

$$\bar{A}^i\overline{B}_j = \frac{\partial \bar{x}^j}{\partial x^m}\frac{\partial x^n}{\partial \bar{x}^j}A^m B_n.$$

Let $\overline{C} = \bar{A}^i\overline{B}_i$, then the contraction by setting $i = j$ and summing all the terms, becomes

$$\begin{aligned}\overline{C} &= \bar{A}^i\overline{B}_i,\\ &= \delta^n{}_m A^m B_n,\\ &= A^n B_n = C.\end{aligned}$$

As we observe, the end result becomes a scalar (as implied by the terminology - scalar product).

□

2.4 Reciprocal Basis and the Metric Tensor

Definition. *Two bases* $(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3)$ *and* $(\mathbf{E}^1, \mathbf{E}^2, \mathbf{E}^3)$ *are said to be reciprocral if they satisfy the condition*

$$E_i E^j = \delta_i{}^j = \begin{cases} 1 & \textit{for } i = j \\ 0 & \textit{for } i \neq j \end{cases}$$

One such basis that satisfies this is

$$E_i = \frac{\partial \mathbf{r}}{\partial u^i}$$

where $\mathbf{r} = x(u,v,w)\mathbf{e}_1 + y(u,v,w)\mathbf{e}_2 + z(u,v,w)\mathbf{e}_3$. The unit vectors e_i are defined as $\frac{E_i}{|E_i|}$. Given the basis $(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3)$, we can always determine $(\mathbf{E}^1, \mathbf{E}^2, \mathbf{E}^3)$ by the following triple scalar products

$$E^i = \frac{e_{ijk}E_j E_k}{e_{ijk}E_i E_j E_k}.$$

Given these new basis vectors, we can now represent any vector, $\mathbf{A}$, by either of the baseis. Given the basis $(\mathbf{E}_1, \mathbf{E}_2, \mathbf{E}_3)$, we can represent $\mathbf{A}$ in the form

$$A^i E_i.$$

The components of $\mathbf{A}$, A^i, relative to basis E_i are called *contravariant components* of $\mathbf{A}$. Similarly, the components A_i relative to the basis E^i are called *covariant components* of $\mathbf{A}$:

$$A_i E^i.$$

The contra and co-variant are different ways of representing $\mathbf{A}$ with respect to a set of reciprocal basis vectors. The inter-relationship between these components is given by the *metric* and the *cojugate metric* of space

$$\begin{aligned} g_{ij} &= E_i E_j = \frac{\partial \mathbf{r}}{\partial u^i} \cdot \frac{\partial \mathbf{r}}{\partial u^j}, \\ g^{ij} &= E^i E^j = (g_{ij})^{-1}, \end{aligned}$$

where the last identity follows from the reciprocity. In index notation, the relationship expressed with the metric becomes

$$A_i = g_{ij}A^j,$$
$$A^i = g^{ij}A_j.$$

Hence, we can both raise the indices, and lower them, by applying either the conjugate metric g^{ij} or the metric g_{ij}. This is an useful operation (e.g. when applied to higher-order tensors like the Riemann curvature tensor.)

Let $x = x(u, v, w), y = y(u, v, w), z = z(u, v, w)$, then the curve element ds is given as

$$ds^2 = g_{ij}du_idu_j.$$

An example of this is illustrated by transforming the Cartesian coordinates (x, y, z) to cylindrical (r, θ, z). The relationship between these coordinate systems is as following

$$x = r\cos(\theta),$$
$$y = r\sin(\theta),$$
$$z = z.$$

Hence, x, y, and z are functions of $u_i = (r, \theta, z)$, and the line vector $\mathbf{r}$ is given as $\mathbf{r} = x(r, \theta, z)\mathbf{e}_1 + y(r, \theta, z)\mathbf{e}_2 + z(r, \theta, z)\mathbf{e}_3$. The the non-zero entries of the metric are the diagonal entries

$$g_{11} = \frac{\partial \mathbf{r}}{\partial r} \cdot \frac{\partial \mathbf{r}}{\partial r}, \qquad g_{22} = \frac{\partial \mathbf{r}}{\partial \theta} \cdot \frac{\partial \mathbf{r}}{\partial \theta}, \qquad g_{33} = \frac{\partial \mathbf{r}}{\partial z} \cdot \frac{\partial \mathbf{r}}{\partial z}.$$

The non-zero calcuated entries of the metric are given as

$$g_{11} = [\cos(\theta)\mathbf{e}_1 + \sin(\theta)\mathbf{e}_2] \cdot [\cos(\theta)\mathbf{e}_1 + \sin(\theta)\mathbf{e}_2] = 1,$$
$$g_{22} = [-r\sin(\theta)\mathbf{e}_1 + r\cos(\theta)\mathbf{e}_2] \cdot [-r\sin(\theta)\mathbf{e}_1 + r\cos(\theta)\mathbf{e}_2] = r^2,$$
$$g_{33} = \mathbf{e}_3 \cdot \mathbf{e}_3 = 1.$$

The line element ds thus becomes

$$ds^2 = dr^2 + r^2d\theta^2 + dz^2.$$

2.5 Tensor Differentiation

Say that we want to differentiate a tensor field $T_{ij}(x^k)$, where $x^k = x^k(\bar{x}^k)$, as following

$$\frac{\partial T_{ij}}{\partial x^k}.$$

We know now that this quantity has to satisfy the transformation law (3). The second order tensor of type $(2, 0)$ satisfies the following transformation law

$$\overline{T}_{pq} = \frac{\partial x^i}{\partial \bar{x}^p}\frac{\partial x^j}{\partial \bar{x}^q}T_{ij}$$

If we now perform differentiation on this expression, we get

$$\frac{\partial \overline{T}_{pq}}{\partial x^k} = \frac{\partial^2 x^i}{\partial x^k \partial \bar{x}^p} \frac{\partial x^j}{\partial \bar{x}^q} T_{ij} + \frac{\partial x^i}{\partial \bar{x}^p} \frac{\partial^2 x^j}{\partial x^k \partial \bar{x}^q} T_{ij} + \frac{\partial x^i}{\partial \bar{x}^p} \frac{\partial x^j}{\partial \bar{x}^q} \frac{\partial T_{ij}}{\partial x^k}.$$

This quantity is certainly not a tensor. In order to derivate a tensor, we need to introduce the Christoffel symbols.

Consider the metric tensor g_{ab} which satifies the transformation law

$$\overline{g}_{kl} = g_{ab} \frac{\partial x^a}{\partial \bar{x}^k} \frac{\partial x^b}{\partial \bar{x}^l}. \tag{10}$$

We define a quantity,

$$(k, l, m) = \frac{\partial \overline{g}_{kl}}{\partial \bar{x}^m}. \tag{11}$$

We insert (10) in (11), and simply apply chain rule

$$(k, l, m) = \frac{\partial g_{ab}}{\partial x^c} \frac{\partial x^c}{\partial x^m} \frac{\partial x^a}{\partial \bar{x}^k} \frac{\partial x^b}{\partial \bar{x}^l} + g_{ab} \frac{\partial^2 x^a}{\partial x^m \partial \bar{x}^k} \frac{\partial x^b}{\partial \bar{x}^l} + g_{ab} \frac{\partial x^a}{\partial \bar{x}^k} \frac{\partial^2 x^b}{\partial x^m \partial \bar{x}^l}.$$

By combining the terms, [Hei01],

$$\frac{1}{2} \left[(k, l, m) + (l, m, k) - (m, k, l) \right]$$

we can extract Christoffel symbol of *first kind*, which is defined as

$$\frac{1}{2} \left[\frac{\partial g_{ab}}{\partial x^c} + \frac{\partial g_{bc}}{\partial x^a} - \frac{\partial g_{ac}}{\partial x^b} \right]. \tag{12}$$

By introducing a shorter notation [Hei01], we can write (12) as

$$[ac, b] = [ca, b].$$

This implies that we have symmetry about the variables a and c. This new quantity is not a tensor, as it does not satisfy the transformation law.

Christoffel symbol of *second kind* is defined as

$$g^{mb} [ac, b] = \frac{1}{2} g^{mb} \left[\frac{\partial g_{ab}}{\partial x^c} + \frac{\partial g_{bc}}{\partial x^a} - \frac{\partial g_{ac}}{\partial x^b} \right].$$

We can write this in a shorter notation by using brackets

$$\begin{Bmatrix} m \\ b\ c \end{Bmatrix} = \begin{Bmatrix} m \\ c\ b \end{Bmatrix} = g^{mb} [ac, b], \tag{13}$$

where we have symmetry between b and c. This quantity is not a tensor either; agian, it does not obey the transformation law. We can interchange between the second kind and first kind, by multiplying (13) with the metric g_{mb}. As $g_{mb}\, g^{mb} = \delta_m{}^b$ (see Section 2.4), we end up with the Christoffel symbol of first kind, (12).

So far we have introduced new notation for derivating tensors, yet the operators themselves are not tensors[1]. The purpose of Christoffel symbols becomes quite apparent, when we use these operators to define a covariant derivative of a tensor.

The covariant derivative of a covariant tensor, A_m, is given as

$$A_{b,c} = \frac{\partial A_b}{\partial x^c} - \begin{Bmatrix} m \\ b\ c \end{Bmatrix} A_m.$$

which becomes a second order tensor, satisfying the transformation law

$$\bar{A}_{i,j} = A_{b,c} \frac{\partial x^b}{\partial \bar{x}^i} \frac{\partial x^c}{\partial \bar{x}^j}.$$

We have sucessfully managed to derivate a tensor, and the resulting operation adheres to the transformation law. In other words, the derivative of the tensor results in a another tensor.

Similary, we can show that the covariant derivative of the contravariant tensor A^m,

$$A^m{}_{,n} = \frac{\partial A^m}{\partial x^n} + \begin{Bmatrix} m \\ l\ n \end{Bmatrix} A^l$$

obeys the transformation law. Here, the first term on the right hand side of the equation is the rate of the tensor field as we move along a coordinate curve, while the second term is the change in local basis vectors as we move along the coordinate curves, [Hei01].

For second order tensors $A_{ij}, A^i{}_j, A^{ij}$, their covariant derivatives are given as

$$\begin{aligned}
A_{ij,k} &= \frac{\partial A_{ij}}{\partial x^k} - A_{mj} \begin{Bmatrix} m \\ i\ k \end{Bmatrix} - A_{im} \begin{Bmatrix} m \\ j\ k \end{Bmatrix}, \\
A^i{}_{j,k} &= \frac{\partial A^i{}_j}{\partial x^k} + A^m{}_j \begin{Bmatrix} i \\ m\ k \end{Bmatrix} - A^i{}_m \begin{Bmatrix} m \\ j\ k \end{Bmatrix}, \\
A^{ij}{}_{,k} &= \frac{\partial A^{ij}}{\partial x^k} + A^{mj} \begin{Bmatrix} i \\ m\ k \end{Bmatrix} + A^{im} \begin{Bmatrix} j \\ m\ k \end{Bmatrix}.
\end{aligned}$$

Given two tensors, say A_{ij} and B_{ij}, the covariant derivation is same as ordinary derivation, where

1. $(A_{ij} + B_{ij})_{,k} = A_{ij,k} + B_{ij,k}$ (derivative of sum is the sum of derivatives)
2. $(A_{ij}B_{ij})_{,k} = A_{ij,k}B_{ij} + A_{ij}B_{ij,k}$ (product rule)
3. $(A_{ij,k})_{,l} = A_{ij,kl}$ (higher-order derivatives are defined as derivatives) [2]

[1] [Hei01] purposely introduced the above notation as to clearly avoid any confusion between a tensor and Christoffel symbols. Another oft-used symbols for Christoffel symbol of second kind is $\begin{Bmatrix} i \\ j\ k \end{Bmatrix} = \Gamma^i_{jk}$. This notation is used for instance by [LR89], and many other authors.

[2] It is worth noting that unlike partial derivatives, where for example the second order derivative of a function f, $\frac{\partial^2 f}{\partial x^i \partial x^j} = \frac{\partial^2 f}{\partial x^j \partial x^i}$. This does not necessarily apply for higher-order covariate derivatives of tensors, where in general $A_{i,jk} \neq A_{i,kj}$.

We can now finally introduce the Riemann-Christoffel Tensor. The tensor is found by the following identity

$$A_{i,jk} - A_{i,kj} = A_m R^m{}_{ijk}, \tag{14}$$

where the fourth-order tensor

$$R^m{}_{ijk} = \frac{\partial}{\partial x^j}\begin{Bmatrix} m \\ i\ k \end{Bmatrix} - \frac{\partial}{\partial x^k}\begin{Bmatrix} m \\ i\ j \end{Bmatrix} + \begin{Bmatrix} n \\ i\ k \end{Bmatrix}\begin{Bmatrix} m \\ n\ j \end{Bmatrix} - \begin{Bmatrix} n \\ i\ j \end{Bmatrix}\begin{Bmatrix} m \\ n\ k \end{Bmatrix}, \tag{15}$$

is called the Riemann-Christoffel tensor[1,2], Riemann curvature tensor, or simply Riemann tensor. The covariant form of the tensor can be found by multiplying the Riemann curvature tensor with the metric g_{pm}

$$R_{pijk} = g_{pm} R^m{}_{ijk}.$$

This tensor is skew-symmetric in two of it's indices

$$\begin{aligned} R_{pijk} &= -R_{pikj} \\ R_{jkpi} &= -R_{jkip} \end{aligned}$$

From the covariant Riemann-Christoffel tensor it follows that there are $N = \frac{1}{12}n^2(n^2-1)$ independent components. For two-dimensions there is only one independent component, while for three-dimensions there are 6 independent components

The tensor therefore provides a way for determining if the space is flat or curved. If the Riemann tensor for a given spacetime is zero every where, initially any parallel geodesics remain parallel (the spacetime is flat, otherwise the spacetime is curved).

We may also define the Ricci tensor R_{jk} by contracting the curvature tensor (15),

$$R_{jk} = R^m{}_{mjk} = g^{pi} R_{pijk}.$$

When expressed with Christoffel symbols, we can show that the Ricci tensor is symmetric. Performing another contraction, we get the scalar curvature

$$R = R^j{}_k = g^{kj} R_{jk}.$$

Since this quantity is invariant in any coordinate system, it gives a coordinate independent measure of a space curvature. However, even though both the Ricci tensor and the scalar curvature can be zero in *curved space*, a non-zero value clearly indicates that the space is curved. Which in it self is useful. Only by evaluating the Riemann tensor can one conclusively distinguish flat and curved space [Moo10].

[1] Named after Bernhard Riemann and Elwin Bruno Christoffel.

[2] Another way to introduce the Riemann curvature tensor is by using the geodesic differential equations. We will later introduce these when we discuss techniques for visualization of second order tensor fields.

2.6 Example: Riemann Curvature Tensor for Toroidal Coordinates

This coordinate system (η, θ, ψ) results from rotating a two-dimensional bipolar coordinate system about the axis that separates it's two foci. The coordinate relationship between toroidal and cartesian coordinate system is given as

$$x = \frac{a\sinh(\eta)\cos(\psi)}{\cosh(\eta) - \cos(\theta)} \tag{16}$$

$$y = \frac{a\sinh(\eta)\sin(\psi)}{\cosh(\eta) - \cos(\theta)} \tag{17}$$

$$z = \frac{a\sin(\theta)}{\cosh(\eta) - \cos(\theta)} \tag{18}$$

where θ coordinate of a point P equals the angle F_1PF_2, where F_1 and F_2 are the two foci as displayed below.

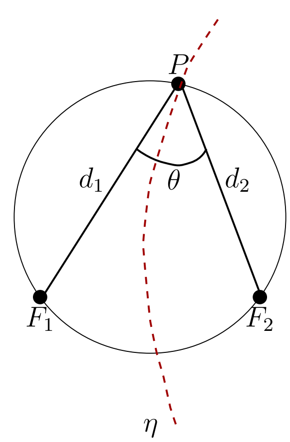


Figure 2: Reference circle on a toroid.

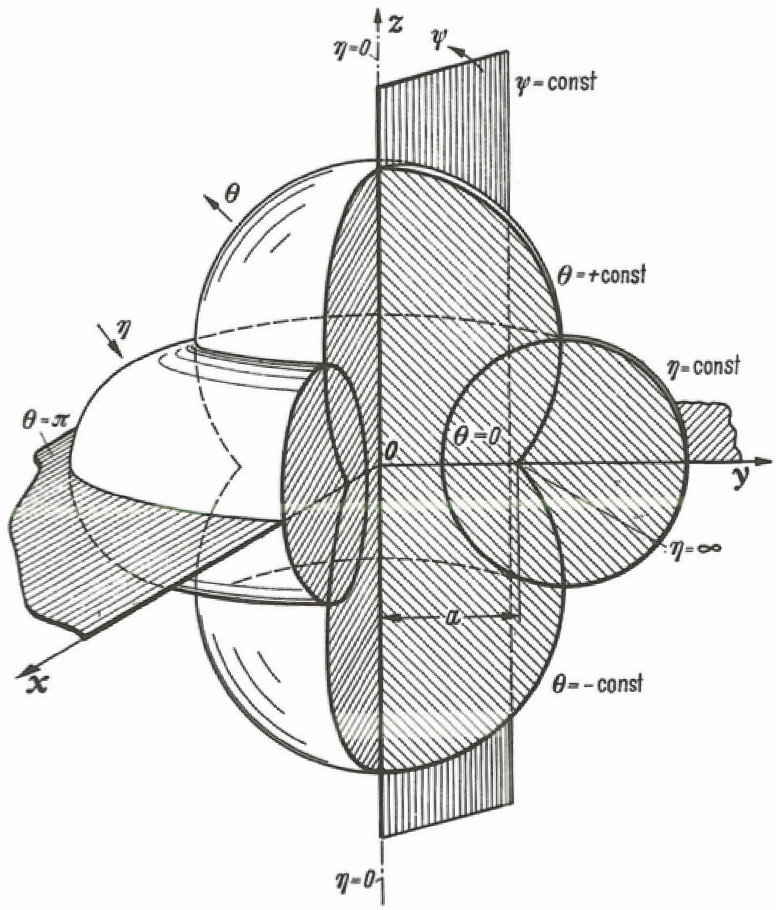


Fig. 4.04. Toroidal coordinates (η, θ, ψ). Coordinate surfaces are toroids ($\eta = \text{const}$), spherical bowls ($\theta = \text{const}$), and half-planes ($\psi = \text{const}$)

Figure 3: Coordinate surfaces for toriodal coordinates. Picture scanned from [MS71].

In the xy-plane the focal ring(also known as the *reference circle*) has radius a. The η coordinate is given as the following ratio

$$\eta = \ln \frac{d_1}{d_2}.$$

The coordinates vary as following :

$$\begin{cases} -\pi & < & \theta & < & \pi, \\ 0 & \leq & \psi & < & 2\pi, \\ & & \eta & \geq & 0. \end{cases}$$

Our first task is to determine the metric (found from the transformation between cartesian and toroidal coordinates), which is given as

$$g_{ij} = \frac{\partial \mathbf{r}}{\partial u_i} \cdot \frac{\partial \mathbf{r}}{\partial u_j},$$

where $u_i = (\eta, \theta, \psi)$, for $i = 1, 2, 3$, and $\mathbf{r}$ is the vector

$$\mathbf{r} = x\hat{\mathbf{e}}_1 + y\hat{\mathbf{e}}_2 + z\hat{\mathbf{e}}_3 = x(\eta, \theta, \psi)\hat{\mathbf{e}}_1 + y(\eta, \theta, \psi)\hat{\mathbf{e}}_2 + z(\eta, \theta, \psi)\hat{\mathbf{e}}_3$$

spanned by cartesian orthogonal basis[1]. We can find all the components of the metric by hand calculations, or through Python using the symbolic package SymPy (see Appendix C.3).

The entries of the metric g_{ij} are found from change of coordinates when applying the equations (16)-(18). The only non-zero entries are found to be the diagonal entries, i.e $i = j$. Which are given as

$$\begin{aligned}
g_{11} &= \frac{\partial \mathbf{r}}{\partial u_1} \cdot \frac{\partial \mathbf{r}}{\partial u_1} = \frac{\partial \mathbf{r}}{\partial \eta} \cdot \frac{\partial \mathbf{r}}{\partial \eta} \\
&= \frac{\partial (x\hat{\mathbf{e}}_1 + y\hat{\mathbf{e}}_2 + z\hat{\mathbf{e}}_3)}{\partial \eta} \cdot \frac{\partial (x\hat{\mathbf{e}}_1 + y\hat{\mathbf{e}}_2 + z\hat{\mathbf{e}}_3)}{\partial \eta} \\
&= \left[\frac{\partial x}{\partial \eta}\right]^2 + \left[\frac{\partial y}{\partial \eta}\right]^2 + \left[\frac{\partial z}{\partial \eta}\right]^2 .
\end{aligned}$$

Similarly, the other non-zero entries of the metric are

$$\begin{aligned}
g_{22} &= \left[\frac{\partial x}{\partial \theta}\right]^2 + \left[\frac{\partial y}{\partial \theta}\right]^2 + \left[\frac{\partial z}{\partial \theta}\right]^2 , \\
g_{33} &= \left[\frac{\partial x}{\partial \psi}\right]^2 + \left[\frac{\partial y}{\partial \psi}\right]^2 + \left[\frac{\partial z}{\partial \psi}\right]^2 .
\end{aligned}$$

Here we list the calculations necessary to find g_{11}. The other entries of g_{ij} are found by using the same techniques as demonstrated here.

$$\begin{aligned}
\frac{\partial x}{\partial \eta} &= \frac{a\cos(\psi)[1 - \cosh(\eta)\cos(\theta)]}{[\cosh(\eta) - \cos(\theta)]^2} \\
&\Downarrow \\
\left[\frac{\partial x}{\partial \eta}\right]^2 &= \frac{a^2\cos^2(\psi)[1 - \cosh(\eta)\cos(\theta)]^2}{[\cosh(\eta) - \cos(\theta)]^4}
\end{aligned}$$

Similary, we find the other expressions

$$\begin{aligned}
\left[\frac{\partial y}{\partial \eta}\right]^2 &= \frac{a^2\sin^2(\psi)[1 - \cosh(\eta)\cos(\theta)]^2}{[\cosh(\eta) - \cos(\theta)]^4} , \\
\left[\frac{\partial z}{\partial \eta}\right]^2 &= \frac{a^2\sinh^2(\eta)\sin^2(\theta)}{[\cosh(\eta) - \cos(\theta)]^4} .
\end{aligned}$$

[1]Which has the property $\hat{e}_i \cdot \hat{e}_j = \delta_{ij}$, and any derivation of the basis is identically zero. Which becomes quite practical when attempting to find the metric.

Adding all the terms together,

$$\begin{aligned}
g_{11} &= \frac{a^2\cos^2(\psi)[1-\cosh(\eta)\cos(\theta)]^2 + a^2\sin^2(\psi)[1-\cosh(\eta)\cos(\theta)]^2 + a^2\sinh^2(\eta)\sin^2(\theta)}{[\cosh(\eta)-\cos(\theta)]^4} \\
&= a^2\left(\frac{1-2\cosh(\eta)\cos(\theta)+\cosh^2(\eta)\cos^2(\theta)+[1-\cos^2(\theta)]\sinh^2(\eta)}{[\cosh(\eta)-\cos(\theta)]^4}\right) \\
&= a^2\left(\frac{\cosh^2(\eta)-2\cosh(\eta)\cos(\theta)+\cos^2(\theta)}{[\cosh(\eta)-\cos(\theta)]^4}\right)
\end{aligned}$$

where we have used the trignometric identities $\sin^2(\psi)+\cos^2(\psi) = \sin^2(\theta)+\cos^2(\theta) = 1$, and $\cosh^2(\eta)-\sinh^2(\eta) = 1$. We can simplify this expression further, by expanding the factor $[\cosh(\eta)-\cos(\theta)]^2 = \cosh^2(\eta)-2\cosh(\eta)\cos(\theta)+\cos^2(\theta)$. Hence, we end up with the first entry of the metric,

$$g_{11} = \frac{a^2}{[\cosh(\eta)-\cos(\theta)]^2}. \tag{19}$$

The other diagonal entries (i.e the non-zero entries of g_{ij}) are found to be

$$g_{22} = g_{11} = \frac{a^2}{[\cosh(\eta)-\cos(\theta)]^2}, \tag{20}$$

$$g_{33} = \frac{a^2\sinh^2(\eta)}{[\cosh(\eta)-\cos(\theta)]^2}. \tag{21}$$

Now we can find the Christoffel symbols for our metric g_{ij}, and thereby be able to determine the Riemann curvature tensor. The process is as following :

1. Determine the non-zero contributions from Equation (12)
2. Determine the non-zero contributions from Equation (13)[1]
3. Finally determine all the derivatives in Equation (15)

Here are points (1) and (2) : In order determine all the terms of the Riemann curvature tensor we must determine the Christoffel symbols of second kind, which again implies that we must determine the corresponding Christoffel symbols of first kind. For the sake of clarity, we restate the Riemann curvature tensor

$$R^m{}_{ijk} = \frac{\partial}{\partial x^j}\begin{Bmatrix} m \\ i\ k \end{Bmatrix} - \frac{\partial}{\partial x^k}\begin{Bmatrix} m \\ i\ j \end{Bmatrix} + \begin{Bmatrix} n \\ i\ k \end{Bmatrix}\begin{Bmatrix} m \\ n\ j \end{Bmatrix} - \begin{Bmatrix} n \\ i\ j \end{Bmatrix}\begin{Bmatrix} m \\ n\ k \end{Bmatrix}, \tag{22}$$

Let us initially focus on the first term here

$$\frac{\partial}{\partial x^j}\begin{Bmatrix} m \\ i\ k \end{Bmatrix}. \tag{23}$$

[1]Notice that the conjugate metric g^{ij} is relatively easy to determine when the metric is diagonal. The entries of the conjugate metric become the inverse of the entries of g_{ij}.

Using the definition for Christoffel symbol of second kind (13), and inserting for the first kind (12), we get the following expression

$$\frac{\partial}{\partial x^j}\begin{Bmatrix} m \\ i\ k \end{Bmatrix} = \frac{\partial}{\partial x^j}\, g^{mk}\,[ai,k] \tag{24}$$

$$= \frac{1}{2}\frac{\partial}{\partial x^j}\, g^{mk}\left(\frac{\partial g_{ak}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^a} - \frac{\partial g_{ai}}{\partial x^k}\right) \tag{25}$$

for $a, i, k, m = 1, 2, 3$, and where the coordinates are $x^i = \eta, \theta, \psi$, for $i = 1, 2, 3$.

There are three cases where we get contribution from the Equation (25).

- Case 1, where $m = k$, and $a = k$,
- Case 2, where $m = k$, and $i = k$,
- Case 3, where $m = k$, and $i = a$.

For every case we require that $m = k$. This is because the conjugate metric g^{mk} is zero every where except on the diagonal. As $g^{mk} = (g_{mk})^{-1}$, we find conjugate metric by inverting the entries of g_{mk}.

For Case 1, we get the following expression

$$\begin{aligned}\frac{1}{2}\frac{\partial}{\partial x^j}\, g^{mk}\left(\frac{\partial g_{ak}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^a} - \frac{\partial g_{ai}}{\partial x^k}\right) &= \frac{1}{2}\frac{\partial}{\partial x^j}\, g^{kk}\left(\frac{\partial g_{kk}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^k} - \frac{\partial g_{ki}}{\partial x^k}\right) \\ &= \frac{1}{2}\frac{\partial}{\partial x^j}\left(g^{kk}\frac{\partial g_{kk}}{\partial x^i}\right)\end{aligned}$$

Similarly, for the other two cases we get the expressions

$$\text{Case 2}: \ m = k, \qquad i = k$$

$$\begin{aligned}\frac{1}{2}\frac{\partial}{\partial x^j}\, g^{mk}\left(\frac{\partial g_{ak}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^a} - \frac{\partial g_{ai}}{\partial x^k}\right) &= \frac{1}{2}\frac{\partial}{\partial x^j}\, g^{kk}\left(\frac{\partial g_{ak}}{\partial x^k} + \frac{\partial g_{kk}}{\partial x^a} - \frac{\partial g_{ak}}{\partial x^k}\right) \\ &= \frac{1}{2}\frac{\partial}{\partial x^j}\left(g^{kk}\frac{\partial g_{kk}}{\partial x^a}\right)\end{aligned}$$

$$\text{Case 3}: \ m = k, \qquad i = a$$

$$\begin{aligned}\frac{1}{2}\frac{\partial}{\partial x^j}\, g^{mk}\left(\frac{\partial g_{ak}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^a} - \frac{\partial g_{ai}}{\partial x^k}\right) &= \frac{1}{2}\frac{\partial}{\partial x^j}\, g^{kk}\left(\frac{\partial g_{ik}}{\partial x^i} + \frac{\partial g_{ki}}{\partial x^i} - \frac{\partial g_{ii}}{\partial x^k}\right) \\ &= \frac{1}{2}\frac{\partial}{\partial x^j}\left(g^{kk}\frac{\partial g_{kk}}{\partial x^i}\right)\end{aligned}$$

Adding the results from all three of the cases, we get the result

$$\frac{\partial}{\partial x^j}\begin{Bmatrix} m \\ i\ k \end{Bmatrix} = \frac{\partial}{\partial x^j}\left(g^{kk}\frac{\partial g_{kk}}{\partial x^i} + \frac{1}{2}g^{kk}\frac{\partial g_{kk}}{\partial x^a}\right) \tag{26}$$

The other terms of the Riemann tensor (22) give corresponding terms

$$-\frac{\partial}{\partial x^k}\begin{Bmatrix} m \\ i\ j \end{Bmatrix} = -\frac{\partial}{\partial x^k}\left(g^{jj}\frac{\partial g_{jj}}{\partial x^i} - \frac{1}{2}g^{jj}\frac{\partial g_{jj}}{\partial x^a}\right) \tag{27}$$

$$\begin{Bmatrix} n \\ i\ k \end{Bmatrix}\begin{Bmatrix} m \\ n\ j \end{Bmatrix} = \left(g^{kk}\frac{\partial g_{kk}}{\partial x^i} + \frac{1}{2}g^{kk}\frac{\partial g_{kk}}{\partial x^a}\right)\left(g^{jj}\frac{\partial g_{jj}}{\partial x^a} + \frac{1}{2}g^{jj}\frac{\partial g_{jj}}{\partial x^i}\right) \tag{28}$$

$$-\begin{Bmatrix} n \\ i\ j \end{Bmatrix}\begin{Bmatrix} m \\ n\ k \end{Bmatrix} = -\left(g^{jj}\frac{\partial g_{jj}}{\partial x^i} + \frac{1}{2}g^{jj}\frac{\partial g_{jj}}{\partial x^a}\right)\left(g^{kk}\frac{\partial g_{kk}}{\partial x^a} + \frac{1}{2}g^{kk}\frac{\partial g_{kk}}{\partial x^j}\right) \tag{29}$$

We see that, for $i = j = k = a = 3$, every term becomes zero (as the respective derivatives are zero). For every other permutation, each term cancels the other, resulting in that every element of the Riemann curvature tensor is zero for the toroidal coordinates. Implying that this is a flat space.

Part II

Method & Results

3 Visualization of Tensor Fields : Current State

Techniques for tensor field visualization can be viewed as two distinct methods : indirect and direct visualization methods. Direct visualization of tensor fields is accomplished with tensor glyphs; ellipsoids that are formed by the eigenvalues of the tensor field, or by drawing integral lines or surfaces. The purpose is to depict full information contained in the field [BH06], [KMW+05], [HFHH04], [FA15]. Indirect methods usually entail the extraction of of spesific structural features from the tensor field, like for example topological features [TS03], [TSH01], [TZP06]. In this thesis, our main focus will be direct visualization methods, with emphasis on integration methods.

3.1 Direct Visualization Methods

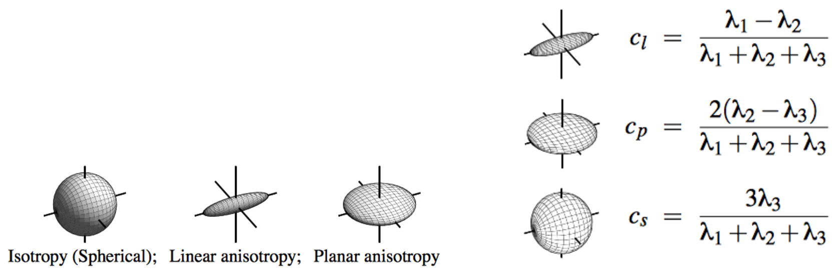
3.1.1 Hyperstreamlines

A second order symmetric tensor in $\mathbf{R}^3$ has six independent elements or components. We can reduce this number down to three by performing an eigendecomposition of the tensor, resulting in three real eigenvalues. Selecting one of these (usually either the major or the minor eigenvalue), we can use it's corresponding eigenvector as the direction for the so-called hyperstreamline trajectories, which in $\mathbf{R}^3$ are tube like structures where the other eigenvalues determine the thickness of the trajectories. They show, for example how forces are transmitted in a stress-tensor field and how momentum is transferred in a momentum-flux-density tensor field. Delmarcelle and Hesselberg, [DH92], were first to introduce the notion of hyperstreamlines. The issue with hyperstreamlines are the same as we find with streamlines for depiction of vector fields, or isosurfaces for scalar fields; occlusion and seeding. In addition, we need to handle degenerate points (similar to the notion of critical points in a vector field, where in the case of tensors, the degenerate points occur when at least two of the eigenvalues are equal.).

In [DH93], Delmarcelle and Hessellink suggest a simply way of handling degenerate points (by simply ignoring them). By assuming that the tensor field is smooth, we can then assume that the direction of the eigenvector is not likely to vary abruptly (i.e vary more than some user defined angle) in between grid points. If, however, the direction changes abruptly, then it is reasonable to assume that the hyperstreamline has crossed a degeneracy. To handle this, we can simply jump over this point and continue integrating along the direction of the eigenvector. Further, we can also keep a track over the transverse eigenvalues to make sure that, where they vanish, we can still include the singularities of the cross-section of the hyperstreamline.

3.1.2 Tensor Glyphs

The traditional approach to direct visualization of symmetric second order tensor fields in $\mathbf{R}^2$ and $\mathbf{R}^3$ have been so-called tensor glyphs. This technique has been employed heavily for medical imaging, depicting e.g. diffusion tensor. Diffusion tensor imaging (DTI) provides information about the diffusion properties of water molecules in brain tissue (which differs for gray matter and white matter). Within white matter the motion of water molecules is restricted in directions that are perpendicular to the fiber tracts. Because of the physical properties of white matter, DTI can be used to investigate white matter in the brain. It's main clinical application has been in the study and treatment of neurological disorders, such as schizophrenia. DTI makes it possible to evaluate the connectivity and coherence of white matter fiber tract, which are tought to be abnormal in schizophrenia [KMW+05]. The diffusion tensor is a second order symmetric tensor in $\mathbf{R}^3$. The tensor is visualized as an ellipsoid, with the longest axis pointing toward the principal direction, i.e the major eigenvector. The shape of the tensor depends upon the strength of the diffusion along the three eigenvectors.



(a) Three basic diffusion tensor shapes.

(b) Linear, planar and isotropic shapes defined by eigenvalues.

Figure 4: Linear, planar and isotropic shapes defined by eigenvalues. Picture taken from [Kin04].

3.1.3 Asymmetric tensors fields

We can always describe any (of either covariant or contravariant type) second order tensor by the sum of their symmetric and anti-symmetric components. For a second-order tensor of type (2,0), we can write it as

$$S_{ij} = \frac{1}{2}\left(S_{ij} + S_{ji}\right) + \frac{1}{2}\left(S_{ij} - S_{ji}\right) \tag{30}$$

A physical example for this case is the gradient of a velocity field, $\nabla \mathbf{v}$. If we denote the components of $\mathbf{v}$ as (v_i), then we can write the gradient as

$$\nabla \mathbf{v} = \begin{bmatrix} \frac{\partial v_1}{\partial x^1} & \frac{\partial v_2}{\partial x^1} & \frac{\partial v_3}{\partial x^1} \\ \\ \frac{\partial v_1}{\partial x^2} & \frac{\partial v_2}{\partial x^2} & \frac{\partial v_3}{\partial x^2} \\ \\ \frac{\partial v_1}{\partial x^3} & \frac{\partial v_1}{\partial x^3} & \frac{\partial v_3}{\partial x^3} \end{bmatrix}, \tag{31}$$

or using index form we can write the components as $\partial v_i / \partial x^j$. Using the above notation (30), we can decompose (31) as

$$\frac{\partial v_i}{\partial x^j} = \frac{1}{2}\left(\frac{\partial v_i}{\partial x^j} + \frac{\partial v_j}{\partial x^i}\right) + \frac{1}{2}\left(\frac{\partial v_i}{\partial x^j} - \frac{\partial v_j}{\partial x^i}\right)$$

The symmetric part is the sum of the Jacobian and transpose of the Jacobian, also refered to as the *rate of strain tensor*. The diagonal entries are the normal strain rates and the off-diagonal entries are the shear strain rates.

We can associate a vector ω_i,

$$\omega = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \begin{bmatrix} \frac{\partial v_3}{\partial x^2} - \frac{\partial v_2}{\partial x^3} \\ \frac{\partial v_1}{\partial x^3} - \frac{\partial v_3}{\partial x^1} \\ \frac{\partial v_2}{\partial x^1} - \frac{\partial v_1}{\partial x^2} \end{bmatrix}$$

with an anti-symmetric tensor defined by

$$R \equiv \begin{bmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{bmatrix}$$

where they are related as

$$R_{ij} = -e_{ijk}\omega_k,$$
$$\omega_k = -\frac{1}{2}e_{ijk}R_{ij}.$$

R is called the *Rotation tensor*.

We can therefore represent the anti-symmetric part as a vector and thereby employ techniques from vector field visualization. For the symmetric part of the tensor, we can perform an eigendecomposition. Which permits us to represent the tensor by it's eigenvectors and their corresponding eigenvalues. Delmarcelle and Hesselberg [DH92] were first to decompose 2D symmetric tensors of type (2,0) in such a manner.

3.1.4 The Geodesic Differential Equations

In Euclidean space, $\mathbf{R}^3$, the distance between two points (x_1, x_2, x_3) and (y_1, y_2, y_3) is given as $d^2 = |x_i - y_i|^2$. In other words, a straight line that connects the two points. According to general relativity, particles in a gravitation field move along geodesics of space-time described by the metric g_{ij}. This is the shortest path between two points on curved space(e.g space-time continuum). In Eucleadian geometry, initially parallel straight lines remain parallel. Initially parallel geodesics in spacetime do not necessarliy remain parallel (as they follow the spacetime curvature).

In order to determine the geodesics, one must solve the following set of differential equations

$$\frac{d^2x^j}{ds^2} + \left\{ \begin{matrix} j \\ h\ k \end{matrix} \right\} \frac{dx^h}{ds}\frac{dx^k}{ds} = 0, \tag{32}$$

where $\left\{ \begin{smallmatrix} j \\ h\ k \end{smallmatrix} \right\}$ is the Christoffel symbol of second kind. The geodesic solution $x^j(s)$ is a curve defined on the interval $s \in [s_0, s_1]$. Often, these equations can not be solved analytically, and we can only solve them numerically.

The solution is a way of depicting the metric tensor. We can extend this method to include any tensor T_{ij}. However, when solving the geodesic differential equations, one of main issues with this method is that the conjugate metric g^{ij} has to exist. If the metric g_{ij} is singular, then we can not solve the differential equations (32). So, if we are to extend the method of visualization of tensor fields T_{ij} to the concept of a metric, then a singular tensor will not work. The momentum flux density tensor $\rho v_i v_j$ is an important tensor which is singular. Hence, for such a tensor, we can not determine the geodesics.

3.2 Indirect Visualization Methods

[Del94] was first to introduce a topological approach for visualization of tensor fields, as noted by [Tri02]. Delmarcelle introduced the notion of degenerate points, which correspond to critical points in vector fields. The following (based on [Del94]) describes an algorithm for tracking degenerate points.

Given the symmetric second rank tensor

$$T = \begin{bmatrix} T_{11} & T_{12} \\ T_{12} & T_{22} \end{bmatrix},$$

For this tensor field, degenerate points satisfy the condition

$$\begin{cases} \frac{T_{11}-T_{22}}{2} & = 0 \\ T_{12} & = 0 \end{cases} \tag{33}$$

Assume that the tensor components can be expressed in the vicinity of degenerate point $\vec{x}_0 = (x_0, y_0)$ as Taylor expansions that start with homogeneous polynomials for $m > 0$

$$\begin{cases} \frac{T_{11}-T_{22}}{2} & \approx P_m(x - x_0, y - y_0) + \cdots \\ T_{12} & \approx Q_m(x - x_0, y - y_0) + \cdots \end{cases}$$

In tensor fields different types of degenerate points can occur that correspond to different local patterns of neighboring hyperstreamlines. These patterns are determined by the tensor gradients at degenerate point positions.

Let $\vec{x}_0 = (x_0, y_0)$ be an isolated degenerate point. Assuming that the functions $T_{11}(\vec{x}) - T_{22}(\vec{x})$ and $T_{12}(\vec{x})$ are analytic, we can expand tensor components in Taylor series around $\vec{x}_0$. After some simplifications, we end up with the following expression

$$\begin{cases} \frac{T_{11}-T_{22}}{2} & \approx a(x - x_0) + b(y - y_0) + \cdots \\ T_{12} & \approx c(x - x_0) + d(y - y_0) + \cdots \end{cases}$$

where

$$a \equiv \frac{1}{2} \frac{\partial(T_{11} - T_{22})}{\partial x}\bigg|_{x_0} \qquad b \equiv \frac{1}{2} \frac{\partial(T_{11} - T_{22})}{\partial y}\bigg|_{y_0}$$

$$c \equiv \frac{1}{2} \frac{\partial T_{12}}{\partial x}\bigg|_{x_0} \qquad d \equiv \frac{1}{2} \frac{\partial T_{12}}{\partial y}\bigg|_{y_0}$$

An important quantity for characterizing degenerate points is

$$\delta = ad - bc$$

which is invariant under rotation.

Definition (Simple and Multiple Degenerate Points)**.** *Let $\vec{x}_0$ be an isolated degenerate point of a tensor field $T \in C^1(E)$, where E is an open subset of $\mathbf{R}^2$, and let $\delta = ad - bc$ be the corresponding third-order invariant. Then, $\vec{x}_0$ is*

- *a simple degenerate point iff* $\delta \neq 0$, *or*
- *a multiple degenerate point iff* $\delta = 0$.

Theorem. *The angle* θ_k *between the x-axis and the sepratrices* s_k *are obtained by computing the real roots* z_k *of the cubic equation*

$$dz^3 + (c + 2b)z^2 + (2a - d)z - c = 0, \tag{34}$$

by inverting the relation $z_k = tan\theta_k$, *and by keeping only those angles that lie along the boundary of a hyperbolic sector.*

The following algorithm handles simple degenerate points only :

1. locate degenerate points by searching for solutions to (33) in every grid cell;
2. classify each degenerate point as a trisector ($\delta < 0$) or a wedge point ($\delta > 0$) by evaluating a,b,c,d and computing $\delta = ad - bc$;
3. select an eigenvector field;
4. solve (34) to find the directions of the three sepratrices (s_1, s_2, s_3) at each trisector point; likewise, extract sepratrices (s_1, s_2) at wedge points where (34) admits three real roots and extract the unique sepratrix at wedge points where (34) has only one real root;
5. integrate hyperstreamlines along the sepratrices; terminate the trajectories wherever they leave the domain, or impinge on the parabolic sector of a wedge point.

3.3 Tensor Overview

Here we provide an overview of second order tensors.

The momentum flux density tensor, stems from the Navier Stokes equations, which we can split into three components

- advection of the i-th component in the j-th direction : $\rho v_i v_j$
- pressure : $p\delta_{ij}$
- stress tensor : σ_{ij}

The stress at a point can completely be specified [KC08] by nine components,

$$\tau = \begin{bmatrix} \sigma_{11} & \tau_{12} & \tau_{13} \\ \tau_{21} & \sigma_{22} & \tau_{23} \\ \tau_{31} & \tau_{32} & \sigma_{33} \end{bmatrix}.$$

Here, we differentiate between the diagonal components; which are normal stresses, and the off-diagonal components; which are shear stresses. The stress tensor describes stresses that act as reaction to external forces. Similarly, the strain tensor is related to the deformation of a body due to stress.

Second order tensor overview					
Tensor	Symmetric	Hyperstremlines	Tensor Glyphs	Geodesics	Application
Velocity gradient	No				Fluid mechanics
Strain rate tensor	Yes	✓	✓		Fluid mechanics
Stress tensor	Yes	✓	✓		Continuum mechanics
Diffusion tensor	Yes	✓	✓	✓	Medical imaging
Reynolds stress	Yes	✓	✓		Computational fluid mechanics
Metric tensor	Yes	✓	✓	✓	Differential geometry/ General relativety
Momentum flux density	Yes	✓	✓		Fluid mechanics

Table 1: Tensor visualization options

The strain rate tensor is related to the velocity field $\mathbf{v}$ by

$$S_{ij} = \frac{1}{2}\left(\frac{\partial v_i}{\partial x^i} + \frac{\partial v_j}{\partial xi}\right)$$

Stress and strain along the eigenvectors are called principal stress and principal strain.

Another important tensor can be derived from the convective term of Reynolds averaged Navier Stokes equations, (the derivative of) the average of the products of the fluctuation velocity components, a second order tensor, with components $-\rho\overline{u_i u_j}$

$$\begin{bmatrix} -\rho u_1 u_1 & -\rho u_1 u_2 & -\rho u_1 u_3 \\ -\rho u_2 u_1 & -\rho u_2 u_2 & -\rho u_2 u_3 \\ -\rho u_3 u_1 & -\rho u_3 u_2 & -\rho u_3 u_3 \end{bmatrix}.$$

This is an important tensor in turbulence modeling. It is called the Reynolds stress tensor[1]. The Reynolds stress tensor is symmetric. The diagonal components are normal stresses, and the off-diagonal components are shear stresses. It is a stress exerted by the turbulenct fluctuations on the mean flow. Another way to interpret Reynolds stress is that it is the rate of mean momentum transfer by turbulent fluctuations [KC08].

In the Table 1 we have listed an overview of different tensors. Note that in order to determine geodesics for a tensor, the Christoffel symbol of second kind has to be invertibel. If a tensor is singular, then this method can not we applied. Similarly, hyperstreamlines are found for symmetric tensors. By decomposing a tensor in symmetric and anti-symmetric components, we can find hyperstreamlines. For the anti-symmetric part of the tensor, we can apply common vector field visualization techniques.

[1]Strictly speaking, the Reynolds stresses are not stresses, but the averaged effect of turbulent convection.

4 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 [CPL+11].

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

4.1 Geodesic Differential Equations

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

$$u'' + \begin{Bmatrix} 0 \\ 0\ 0 \end{Bmatrix} (u')^2 + 2\begin{Bmatrix} 0 \\ 0\ 1 \end{Bmatrix} u'v' + \begin{Bmatrix} 0 \\ 1\ 1 \end{Bmatrix} (v')^2 = 0,$$
$$u'' + \begin{Bmatrix} 1 \\ 0\ 0 \end{Bmatrix} (u')^2 + 2\begin{Bmatrix} 1 \\ 0\ 1 \end{Bmatrix} u'v' + \begin{Bmatrix} 1 \\ 1\ 1 \end{Bmatrix} (v')^2 = 0.$$

where $\left\{ {m \atop i\ j} \right\} = \left\{ {m \atop i\ j} \right\} (u(s), v(s))$ 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 $q = v'$

$$p' + \begin{Bmatrix} 0 \\ 0\ 0 \end{Bmatrix} p^2 + 2\begin{Bmatrix} 0 \\ 0\ 1 \end{Bmatrix} pq + \begin{Bmatrix} 0 \\ 1\ 1 \end{Bmatrix} q^2 = 0,$$
$$q' + \begin{Bmatrix} 1 \\ 0\ 0 \end{Bmatrix} p^2 + 2\begin{Bmatrix} 1 \\ 0\ 1 \end{Bmatrix} pq + \begin{Bmatrix} 1 \\ 1\ 1 \end{Bmatrix} q^2 = 0.$$

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
```

[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

```
        y3 = y[3] # v'
        dy = np.zeros_like(y)
        dy[0] = y1
        dy[2] = y3

        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 C.4.

4.2 The 3D Kerr Metric

Several metric examples are stored in file find_metric.py C.3. 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 $x =$

[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)[Moo10].

$r\sin(\theta)\cos(\phi)$, $y = r\sin(\theta)\sin(\phi)$, $z = r\cos(\theta)$. Then the Kerr metric is given as

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

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)
```

[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 [Moo10].

```
    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.

4.3 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_
```

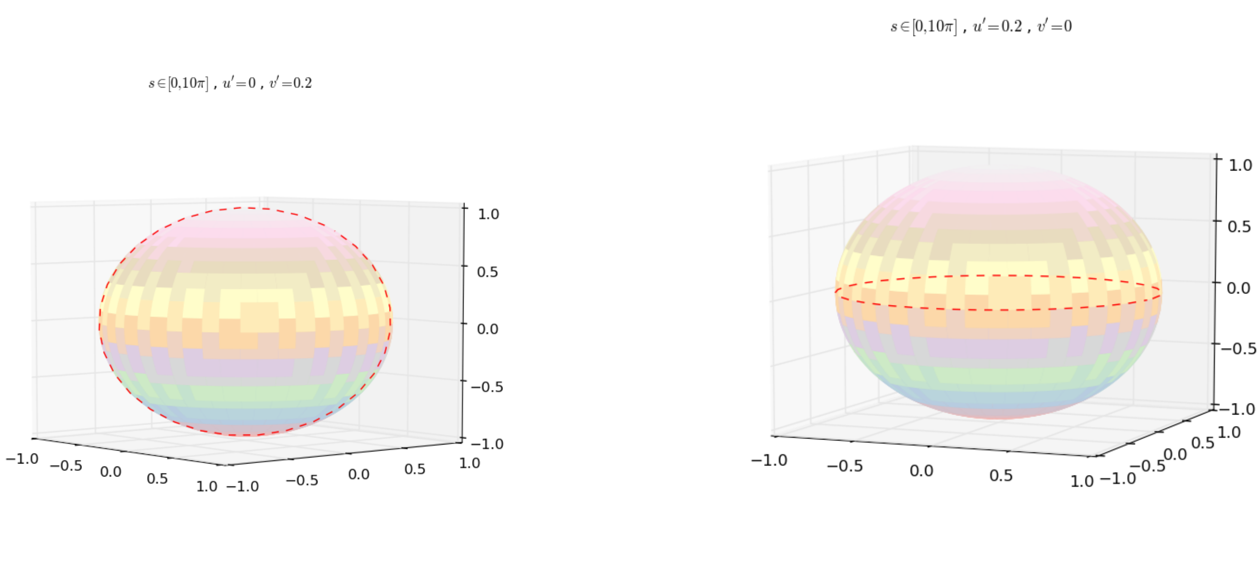
5 Results & Interpretation

5.1 Geodesic Solver

5.1.1 Sphere

The transformation from spherical to cartesian coordinates is given as

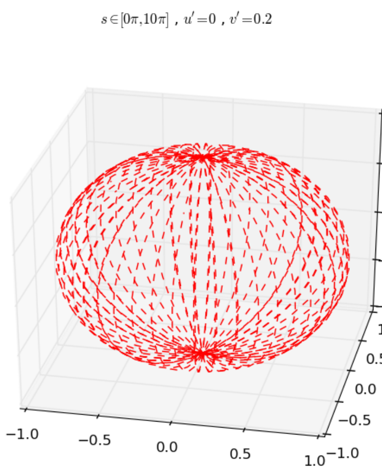
$$
\begin{aligned}
x &= u\sin(v)\cos(w) \\
y &= u\sin(v)\sin(w) \\
z &= u\cos(v)
\end{aligned}
$$



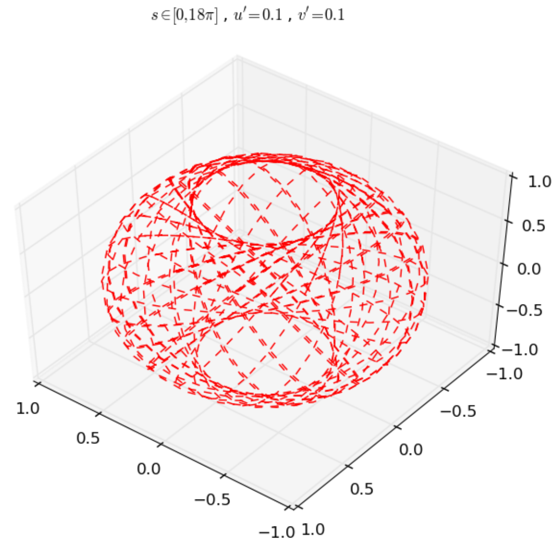
(a) $u' = 0, v' = 0.2$.

(b) $u' = 0.2, v' = 0$.

Figure 5: Geodesic curves on a sphere



(a) $u' = 0, v' = 0.2$.



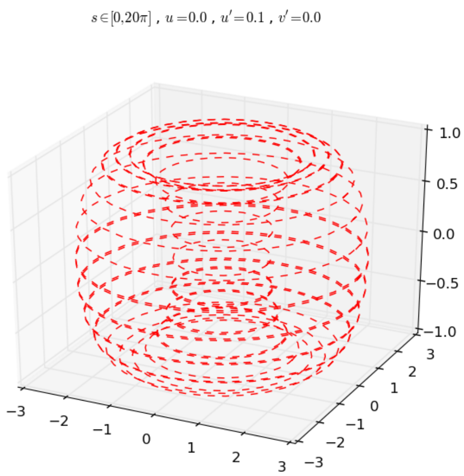
(b) $u' = 0.1, v' = 0.1$.

Figure 6: Geodesic curves on a sphere.

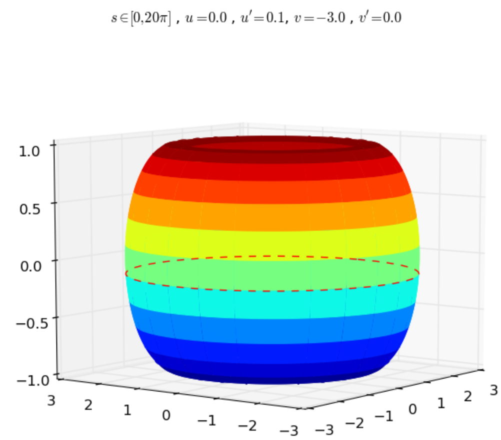
5.1.2 Torus

The transformation from toridal to cartesian coordinates is given as

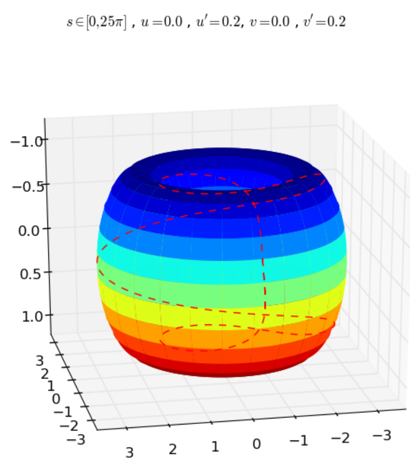
$$
\begin{aligned}
x &= (c + a\cos(u))\cos(v) \\
y &= (c + a\cos(u))\sin(v) \\
z &= \sin(u)
\end{aligned}
$$



(a) $u' = 0.1, v' = .0$.

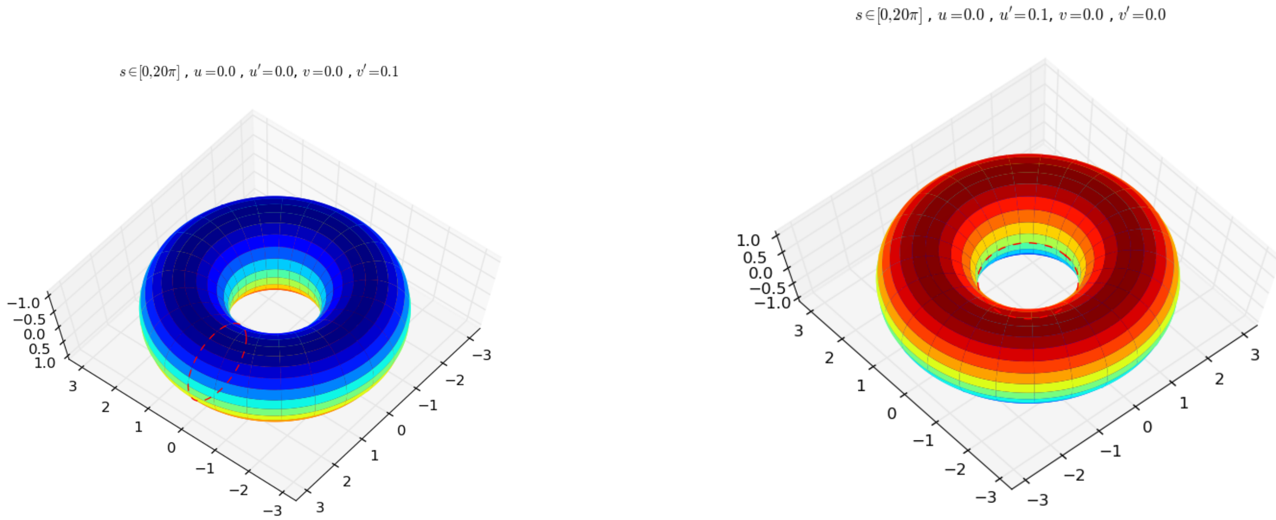


(b) $u' = 0, v' = 0.1$.



(c) $u' = 0.2, v' = 0.2$.

Figure 7: Geodesic curves on a torus.



(a) $u' = 0.1, v' = 0.1$.

(b) $u' = 0.1, v' = 0$.

Figure 8: Geodesic curves on a torus.

5.1.3 Cylindrical Catenoid

$$
\begin{aligned}
x &= \cos(u) - v\sin(u) \\
y &= \sin(u) + v\cos(u) \\
z &= v
\end{aligned}
$$

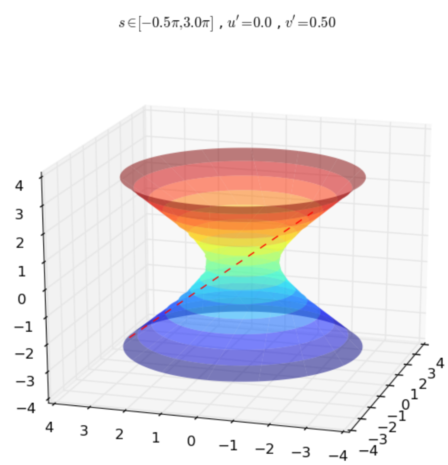
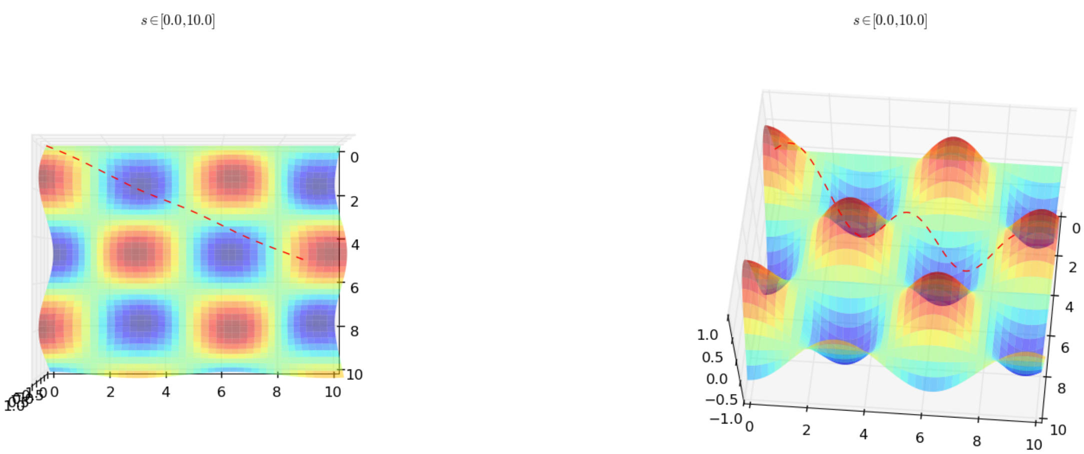


Figure 9: Geodesic curves on a cylindrical catenoid.

5.1.4 Egg Carton Surface

$$
\begin{aligned}
x &= u \\
y &= v \\
z &= \sin(u)\cos(v)
\end{aligned}
$$



(a) $u' = 0.1, v' = 0.1.$

(b) $u' = 0.1, v' = 0.1.$

Figure 10: Geodesic curves on an egg carton surface.

5.1.5 Mobius Strip

$$
\begin{aligned}
x &= \left[1 + \frac{v}{2}\cos\left(\frac{u}{2}\right)\right]\cos(u) \\
y &= \left[1 + \frac{v}{2}\cos\left(\frac{u}{2}\right)\right]\sin(u) \\
z &= \frac{v}{2}\sin\left(\frac{u}{2}\right)
\end{aligned}
$$

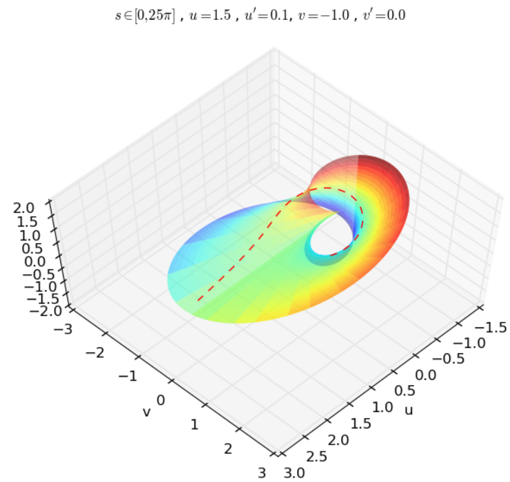


Figure 11: Geodesic curve on a Mobius strip.

5.1.6 3D Kerr Metric

The Kerr metric is given as

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

For $a = 0$, this reduces to the Schwarzchild metric. We have run the geodesic solver for this case, and used the Schwarzchild radius to determine a relationship between the coeffiencts G and M, which amounts to determining the geodesics near a black hole.

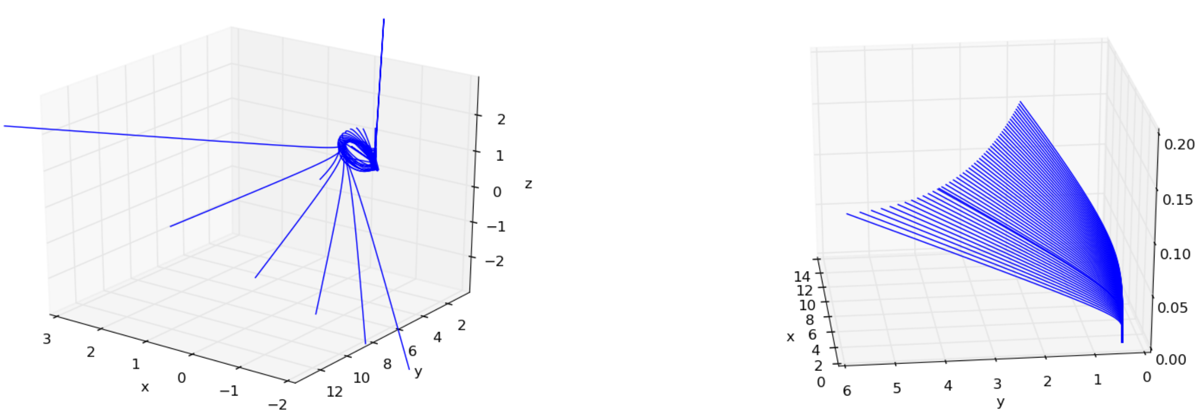


Figure 12: Geodesic curves near a black hole.

6 Conclusion

The software we have developed so far has room for much improvement. Firstly, the hyperstreamline module does not handle closed-hyperstreamlines. This is an important case to handle, as to prevent infinite cycling during hyperstreamline calculation [WM06].Topological features such as degenerate points are also important cases that must be handeled. The topology of a tensor field is the topology of its eigenvectors. In the case of two or more eigenvalues being equal, at least one eigenvector is linearly dependent. As a consequence, hyperstreamlines at such points, can branch out through multiple paths. Finally, perhaps the most important physical characteristic of any tensor field is its time dependency. Handling an unsteady tensor field is important as many physical processes are unsteady.

One of the main challenges of tensor field visualization is the difficulty there lies in adapting same techniques across datasets with different physical attributes. Examples of such fields include stress and strain tensors, rate of deformation tensor, and the diffusion tensor. The physical meaning of tensors can greatly impact how they should be visualized, even when the mathematical representations of these tensors are the same (as we have shown in 2) [HHK+14]. Using differntial geometry, we have demonstrated a new method, where we solve the geodesic differential equations and apply similar techniques as hyperstreamlines. The method itself was applied on metric tensor. It can easily be extended to any other second order tensor. Though, there are limitation, such as the Christoffel symbol of second kind exists only if the metric is non-singular. We showed that the momentum flux density is one such tensor which can not be applied.

We can take the geodesics one step further, if we can manage to combine hyperstreamlines techniques using geodesics to determine the pricipal direction. This hybrid geodesic-hyperstreamline method requires further investigation.

The important finding of this thesis are as following : There is a gaping disparity for readily available free software which permit the user multiple visualization methods for second order (or higher order) tensor fields. As such, we set ourselves upon the daunting task of creating our own tensor module (even though we limited ourselves to a few methods). However, the process it self was quite revelatory. To create a module from almost scratch is an exiciting task, but still quite difficult. As such, much of the focus has been in the implementation process itself.

A Visualization of Vector Fields

A.1 Notation

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

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

$$\mathbf{v} = v_1\hat{e}_1 + v_2\hat{e}_2 + v_3\hat{e}_3. \tag{35}$$

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[Hei01] that we are refering to the component form (35). Likewise, we can define a vector in $\mathbf{R}^N$, where the N components in component form, are given as $(v_1, \ldots, v_N)$. Dots imply the remaining components of the vector, instead of listing them all up. The vector space is spanned by the basis $\{\hat{e}_1, \ldots, \hat{e}_N\}$.

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

$$\|\mathbf{v}\| = \sqrt{v_1^2 + \ldots + v_N^2}\,. \tag{36}$$

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(35). 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(35), we can extend this to N-dimensional vector spaces. The vector components would then be functions of $\mathbf{x} = (x_1, \ldots, x_N)$. A simple vector field (x, y, z) is displayed in Figure 13. 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.

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

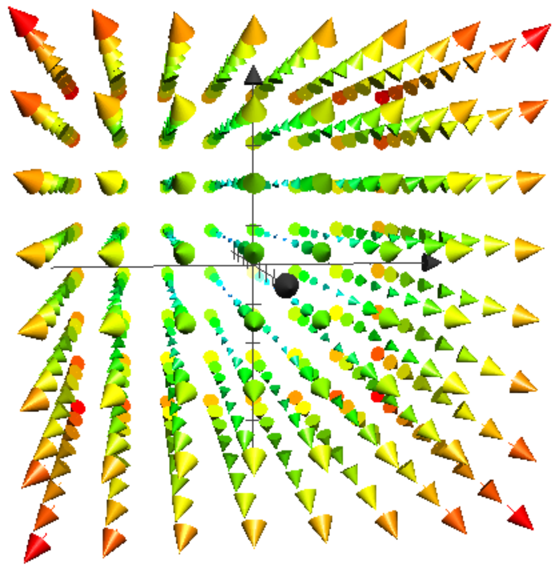


Figure 13: Vector field (x,y,z) displayed using a simple graphical tool in OS X.

A.2 Streamlines

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

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

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

$$\left.\frac{dx_1}{v_1(\mathbf{x})}\right|_{t=t_0} = \left.\frac{dx_2}{v_2(\mathbf{x})}\right|_{t=t_0} = \left.\frac{dx_3}{v_3(\mathbf{x})}\right|_{t=t_0} \tag{37}$$

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.

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

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

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

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

A.3 Numerical Integration

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

Algorithm 1 Integration by forward Euler scheme

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

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

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

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

B Eigendecomposition

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

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

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

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

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

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

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

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

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

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

C Code Listings

C.1 Locate Degenerate Points

degeneracies.py

```
import numpy as np

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

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

def degenerate(T,tol=1e-12):
```

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

```
    """
    Function assumes that the tensor is given with the following form :

    If the tensor values are distributed over a 100x100 grid, then for 2D,
    (100,100,2,2) is the shape of the tensor T. Meaning, each grid point
    contains a 2 dimensional second rank tensor. We assume that all such
    tensors are symmetric. For a 3D tensor defined over a 16x50x16 grid,
    T is of the shape (16,50,16,3,3). Here too we assume that all grid
    tensors are symmetric.
    """
    if T.shape[-2:] == (2,2):
        if len(T.shape[:-2]) == 2:
            deg_points = _find_all_degen_points(T,dim=2,tol=tol)
            return deg_points
    elif T.shape[-2:] == (3,3):
        if len(T.shape[:-2]) == 3:
            deg_points = _find_all_degen_points(T,dim=3,tol=tol)
            return deg_points
    msg = "Tensor has wrong shape.\n"
    msg += "Tensor shape must be either in 3D\n"
    msg += "(data x, data y, data z, gridx, gridy, gridz)\n"
    msg += "or in 2D\n(data x, data y, gridx, gridy)"
    raise IndexError(msg)

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

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

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

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

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

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

```
            T[i,j,k,0,1] == 0
            T[i,j,k,0,2] == 0
            T[i,j,k,1,2] == 0
        """
        found = np.zeros(5)
        for i in range(xdata):
            for j in range(ydata):
                for k in range(zdata):
                    found[0] = np.fabs(T[i,j,k,0,0] - T[i,j,k,1,1]) <= tol
                    found[1] = np.fabs(T[i,j,k,1,1] - T[i,j,k,2,2]) <= tol
                    found[2] = np.fabs(T[i,j,k,0,1]) <= tol
                    found[3] = np.fabs(T[i,j,k,0,2]) <= tol
                    found[4] = np.fabs(T[i,j,k,1,2]) <= tol

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

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

C.2 Invariance

invariance.py

```
import sympy
from sympy import diff
import numpy

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

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

                T = [T11 T12; T12 T22]

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

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

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

```
    T22 = 0.5*T[1,1]

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

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

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

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

C.3 Find the Metric g_{ij}

find_metric.py

```
import sympy as sym

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

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

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

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

    n = ds2
    L='left'
```

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

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


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

```
    if dim == 4:
        t1 = coord1[3]
        t2 = coord2[3]
    dxdu = diff(x,u)
    dxdv = diff(x,v)
    dydu = diff(y,u)
    dydv = diff(y,v)
    if dim >= 3:
        dxdw = diff(x,w)
        dydw = diff(y,w)
        dzdu = diff(z,u)
        dzdv = diff(z,v)
        dzdw = diff(z,w)
    if dim == 4:
        dxdt = diff(x,t2)
        dydt = diff(y,t2)
        dzdt = diff(z,t2)
        dtdu = diff(t1,u)
        dtdv = diff(t1,v)
        dtdw = diff(t1,w)
        dtdt = diff(t1,t2)


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

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

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

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

```

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

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

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

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

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

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

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

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

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

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

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

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

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

def kerr_3D_metric_time_independent(): # unphysical ?
    from sympy import symbols, simplify, cos, sin
    from sympy.abc import G,M,l,u,v,w
    ds2 = '(p/l)*du**2 + p*dv**2 \
           + (u**2 + a**2 + (us*u*a**2*sin(v)**2/p**2))*sin(v)**2*dw**2'
    g = curve_to_metric(ds2,dim=3)
    us,p,a,l = symbols('us,p,a,l')
```

```
    g = g.subs({p:u**2 + a**2*cos(v)})
    g = g.subs({l:u**2 - us*u + a**2})
    g = g.subs({us:2*G*M})
    print "Performing simplification on the metric. This may take some time ...."
    g = simplify(g)
    diff_form = [['du*du','du*dv','du*dw'],
                 ['dv*du','dv*dv','dv*dw'],
                 ['dw*du','dv*dw','dw*dw']]
    return g, diff_form

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

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

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

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

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

    """
    g2,diff_form = cylindrical_coordinates()
```

```
    print "\nCylindrical"
    print g2

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

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

C.4 Riemann Curvature, Ricci tensor, Scalar curvature

tensor.py

```
from __future__ import division
import sympy as sympy

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

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

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

```
            self.t = t

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

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

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

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

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

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

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

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

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

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

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

    def _metric_to_twoform(self,g):
        dim = self.dim
        system = self.system
        diff_forms = system.base_oneforms()
        u_, v_ = self.u, self.v
        u = u_
        v = v_
```

```python
        if dim >= 3:
            w_ = self.w
            w = w_
        if dim == 4:
            t_ = self.t
            t = t_

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

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

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

```
                letter = letter+more
                pos = pos + 1
        for alphabet in abc.__dict__:
            if letter == alphabet:
                letters.append(alphabet)
                self._try_expr(expr[expr.find(alphabet):]) # search for the next unknown variable

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

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

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

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

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

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

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

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

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

        The returned value is a SymPy Matrix.
        """
        from sympy.diffgeom import metric_to_Ricci_components
```

```
        metric = self.metric
        RR = metric_to_Ricci_components(metric)
        simpRR = self._tuple_to_list(RR)
        dim = self.dim
        if form=="simplified":
            print 'Performing simplifications on each component....'
            for m in range(dim):
                for i in range(dim):
                    expr = str(RR[m][i])
                    expr = self._symplify_expr(expr)
                    simpRR[m][i] = expr
        self.Ricci = sympy.Matrix(simpRR)
        return self.Ricci

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


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

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

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

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

```
    print RR
    print "\nand the scalar curvature is"
    print scalarRR


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

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

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

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

C.5 Geodesic Differential Equations Solver

gde.py

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

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

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

```
            -2*C[2,0][2]*dy[0]*dy[4]\
            -2*C[2,1][2]*dy[2]*dy[4]\
              -C[2,1][1]*dy[2]**2\
              -C[2,2][2]*dy[4]**2
    return dy

def f(y,s,*args):
    """
    The geodesic differential equations are solved.
    Described as a system of first order differential-
    equations :

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

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

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

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

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

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

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

```python
        while r.successful() and r.t <= s1:
            r.integrate(r.t + ds)
            y.append(r.y)
        return np.array(y)

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

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

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

def torus(a=1,c=2):
    import find_metric
```

```
    g = find_metric.torus_metric(a,c)
    C = Christoffel_2nd(g)
    return C

def toroid(u=1,v=None,a=1):
    import find_metric as fm
    g, diff = fm.toroidal_coordinates()
    if v is None:
        g = g.subs('u',u)[:2,:2]
    else:
        g = g.subs('v',v)[1:,1:]
    g = g.subs('a',a)

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


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

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

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

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

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

def sphere():
    from sympy.abc import u,v
    from sympy import tan, cos ,sin
    """
    return flat_sphere() # in correct entries in Christoffel symbol of 2nd kind
    """
```

```
    return sym.Matrix([[(0,-tan(v)), (0, 0)],[(sin(v)*cos(v), 0), (0, 0)]])

def mobius_strip():
    import find_metric as fm
    import tensor as t
    g,diff = fm.mobius_strip()
    R = t.Riemann(g,dim=2,sys_title='mobius_strip',flat_diff = diff)
    #metric=R.metric
    from sympy.diffgeom import TensorProduct, Manifold, Patch, CoordSystem
    manifold = Manifold("M",2)
    patch = Patch("P",manifold)
    system = CoordSystem('mobius_strip', patch, ["u", "v"])
    u, v = system.coord_functions()
    du,dv = system.base_oneforms()
    from sympy import cos
    metric = (cos(u/2)**2*v**2/4 + cos(u/2)*v + v**2/16 + 1)*TensorProduct(du, du) + 0.25*TensorProduct(
        dv, dv)
    C = Christoffel_2nd(metric=metric)
    return C

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

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

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

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

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

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

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

```
    u,v = plt.meshgrid(np.linspace(-np.pi,np.pi,150),np.linspace(-np.pi,np.pi,150))
    x = np.cos(u) - v*np.sin(u)
    y = np.sin(u) + v*np.cos(u)
    z = v

    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.view_init(elev=20, azim=-163)
    # use transparent colormap
    import matplotlib.cm as cm
    theCM = cm.get_cmap()
    theCM._init()
    alphas = -.5*np.ones(theCM.N)
    theCM._lut[:-3,-1] = alphas
    ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

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

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

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

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

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

```

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

    import matplotlib.pylab as plt
    from mpl_toolkits.mplot3d import Axes3D
    N = X[:,0].shape[0]
    u,v = plt.meshgrid(np.linspace(0,2*np.pi,250),np.linspace(0,2*np.pi,250))
    x = (c + a*np.cos(v))*np.cos(u)
    y = (c + a*np.cos(v))*np.sin(u)
    z = np.sin(v)

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

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

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

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

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

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

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

```

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

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

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

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

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

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

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

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

    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
```

```
    if multiple is not True:
        X = solve(C,u0,s0,s1,ds,solver)
        r = X[:,0]
        theta = X[:,2]
        # for time independent kerr metric use :
        #phi = X[:,4]
        #x = r*np.sin(theta)*np.cos(phi)
        #y = r*np.sin(theta)*np.sin(phi)
        #z = r*np.cos(theta)
        #ax.plot(x,y,z,'b')
        t = X[:,4]
        x = r*np.sin(theta)
        y = r*np.cos(theta)
        z = t
        ax.plot(x,y,z,'b')

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

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

    return X,plt

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

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

```

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

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

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

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

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

```python
        theCM = 'Pastel1'
        ax.plot_surface(x,y,z,linewidth=0,cmap=theCM)
    plt.hold('on')

    if surface == 'catenoid':
        for u_val in u0_range:
            u0[3] = u_val
            plt = display_multiple_catenoid(u0,s0,s1,ds,C,u0_range)
    elif surface == 'egg_carton':
        for u_val in u0_range:
            u0[0] = u_val
            plt = display_multiple_egg_carton(u0,s0,s1,ds,C,u0_range)
    elif surface == 'sphere':
        for u_val in u0_range:
            u0[0] = u_val
            plt = display_multiple_sphere(u0,s0,s1,ds,C,u0_range)
    elif surface == 'torus':
        for u_val in u0_range:
            u0[0] = u_val # alternate v0 values
            plt = display_multiple_torus(u0,s0,s1,ds,C,u0_range)


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

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

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

        u0 = [0,.2,0,.2] # u(s0), u'(s0), v(s0), v'(s0)
        s0 = 0
        s1 = 25*pi
        ds = .1
```

```
        a = 1
        c = -2
        display_torus(u0,s0,s1,ds,a=a,c=c,show=True)
        #display_multiple_geodesics(u0,s0,s1,ds,'torus',with_object=False)

        u0 = [0,.5,0.5,sqrt(3)/2] # u(0), u'(0), v(0), v'(0)
        s0 = -pi
        s1 = pi
        ds = 0.05
        display_egg_carton(u0,s0,s1,ds,show=True)
        #display_multiple_geodesics(u0,s0,s1,ds,'egg_carton')

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


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

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

C.6 Hyperstreamlines

hyperstreamlines.py

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

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

```python

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

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

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

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


def _expand_hyperstreamline3D(p,Ux,Uy,Uz,Ux_,Uy_,Uz_,l,l_,points):
    p_ = np.zeros_like(p)
```

```
    def f1(x):
        return [griddata(points,Ux,x)[0], griddata(points,Uy,x)[0],griddata(points,Uz,x)[0]]
    def f2(x):
        return [griddata(points,Ux_,x)[0], griddata(points,Uy_,x)[0],griddata(points,Uz_,x)[0]]
    return p_

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

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

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

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

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

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

def _run_example_flat_sphere(xstart,xend,N,direction='major',solver=None):
    """
    A test example, using the metric of a flat sphere, to calculate hyperstreamlines
```

```
    for a 2D grid.
    """
    x0,y0 = xstart
    xN,yN = xend
    Nx,Ny = N
    x,y = np.mgrid[x0:xN:Nx*1j,y0:yN:Ny*1j]
    # Initialize the metric for the flat sphere
    g = np.array([[1,0],[0,1]],dtype=np.float32)
    T = np.zeros([2,2,Nx,Ny],dtype=np.float32) # The tensor field
    eig_field = np.zeros([3,2,Nx,Ny],dtype=np.float32) # The "eigen" field

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

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

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

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

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

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

    if len(sys.argv) > 1:
```

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

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

References

[BH06] W. Benger and H.-C. Hege. Strategies for Direct Visualization of Second-Rank Tensor Fields. In *Visualization and Processing of Tensor Fields*, pages 191–214. Springer-Verlag, 2006.

[CL93] B. Cabral and L. C. Leedom. Imaging Vector Fields Using Line Integral Convolution. *Computer Graphics and Applications*, pages 263–270, 1993.

[CPL$^+$11] G. Chen, D. Palke, Z. Lin, H. Yeh, P. Vincent, R.S. Laramee, and E. Zhang. Asymmetric Tensor Field Visualization for Surfaces. *Visualization and Computer Graphics*, Vol. 17, Issue 12:1979–1988, December 2011.

[Del94] T. Delmarcelle. *The Visualization of Second-Order Tensor Fields*. PhD thesis, Stanford University, 1994.

[DH92] T. Delmarcelle and L. Hesselink. Visualizing Second Order Tensor Fields and Matrix. *Visualization '92 Proceedings*, pages 316–323, 1992.

[DH93] T. Delmarcelle and L. Hesselink. Visualizing Second-Order Tensor Fields with Hyperstreamlines. *Computer Graphics and Applications*, 13(4):25–33, 1993.

[FA15] F. Fu and N. M. Abukhdeir. A Topologically-Informed Hyperstreamline Seeding Method for Alignment Tensor Fields. *Visualization and Computer Graphics*, Vol. 21, Issue 3:413–419, March 2015.

[Hei01] J. H. Heinbockel. *Introduction to Tensor Calculus and Continuum Mechanics*. Trafford Publishing, 2001.

[HFHH04] I. Hotz, L. Feng, H. Hagen, and B. Hamann. Physically Based Methods for Tensor Fields Visualization. *IEEE Visualization*, pages 123–130, 2004.

[HHK$^+$14] M. Hlawitschka, I. Hotz, A. Kratz, G. E. Marai, R. Moreno, G. Scheuermann, M. Stommel, A. Wiebel, and E. Zhang. Top Challenges in the Visualization of Engineering Tensor Fields. In *Visualization and Processing of Tensors and Higher Order Descriptors for Multi-Valued Data*, pages 3–15. Springer-Verlag, 2014.

[KC08] P. L. Kundu and I. M. Cohen. *Fluid Mechanics*. Elsevier, 4 edition, 2008.

[Kin04] G. L. Kindlmann. Superquadric Tensor Glyphs. *Joint Eurographics and IEEE Symposium on Visualization*, pages 147–154, 2004.

[KMW$^+$05] M. Kubicki, R. McCarley, C.-F. Westin, H.-J Park, R. Kikinis S. Maier, F. A. Jolesz, and M. E. Shenton. A Review of Diffusion Tensor Imaging Studies in Schizophrenia. *Journal of Psyhiatric Research*, 41(1-2):15–30, 2005.

[LR89] D. Lovelock and H. Rund. *Tensors, Differential Forms, and Variational Principles*. Dover, 1989.

[Moo10] T. A. Moore. *A General Relativity Workbook*. β0.92 edition, 2010.

[MS71] P. Moon and D. E. Spencer. *Field Theory Handbook, 2.edition*. Springer Verlag, 1971.

[Tri02] X. Trichoche. *Vector and Tensor Field Topology Simplification, Tracking, and Visualization*. PhD thesis, University of Kaiserslautern, 2002.

[TS03] X. Tricoche and G. Scheuermann. Topology Simplification of Symmetric, Second-Order 2D Tensor Field. In *Geometric Modeling for Scientific Visualization*, pages 275–291. Springer-Verlag, 2003.

[TSH01] X. Tricoche, G. Scheuermann, and H. Hagen. Tensor Topology Tracking: A Visualization Method for Time Dependent 2D Symmetric Tensor Fields. *Computer Graphics Forum*, 20(3):461–470, 2001.

[TZP06] X. Tricoche, X. Zheng, and A. Pang. Vsualizing the Topology of Symmetric, Second-Order, Time-Varying Two-Dimensional Tensor Fields. In *Visualization and Processing of Tensor Fields*, pages 225–240. Springer-Verlag, 2006.

[WM06] T. Wischgoll and J. Meyer. Locating Closed Hyperstreamlines in Second Order Tensor Fields. In *Visualization and Processing of Tensor Fields*, pages 257–267. Springer-Verlag, 2006.