**SCHOOL *of*
GRADUATE STUDIES**
EAST TENNESSEE STATE UNIVERSITY

**East Tennessee State University
Digital Commons @ East
Tennessee State University**

Electronic Theses and Dissertations

Student Works

8-2016

An Algorithm for the Machine Calculation of Minimal Paths

Robert Whitinger
East Tennessee State University

Follow this and additional works at: https://dc.etsu.edu/etd

🎯 Part of the Analysis Commons, Numerical Analysis and Computation Commons, Numerical Analysis and Scientific Computing Commons, Other Mathematics Commons, Partial Differential Equations Commons, and the Theory and Algorithms Commons

An Algorithm for the Machine Calculation of Minimal Paths

————————

A thesis

presented to

the faculty of the Department of Mathematics

East Tennessee State University

In partial fulfillment

of the requirements for the degree

Master of Science in Mathematical Sciences

————————

by

Robert James Whitinger

August 2016

————————

Jeff Knisley, Ph.D., Chair

Ariel Cintron-Arias, Ph.D.

Robert Gardner, Ph.D.

Keywords: differential geometry, calculus of variations, functional analysis,
numerical approximation, minimal path.

ABSTRACT

An Algorithm for the Machine Calculation of Minimal Paths

by

Robert James Whitinger

Problems involving the minimization of functionals date back to antiquity. The mathematics of the calculus of variations has provided a framework for the analytical solution of a limited class of such problems. This paper describes a numerical approximation technique for obtaining machine solutions to minimal path problems. It is shown that this technique is applicable not only to the common case of finding geodesics on parameterized surfaces in $\mathbb{R}^3$, but also to the general case of finding minimal functionals on hypersurfaces in $\mathbb{R}^n$ associated with an arbitrary metric.

DEDICATION

Dedicated to my father, James Darwin Whitinger (1925-2011), the man who taught me to love learning and to strive to know the unknown.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

10

1 INTRODUCTION

We notice that nature often "chooses" paths of minimum action. For this reason, many problems in the sciences express themselves in terms of minimal paths. Only a relatively small number of these problems lend themselves to a detailed analytical solution. To counter this deficiency many problem domains are using simple and ideal models. Many problem domains could benefit from more advanced and detailed models if solutions in such models could be found.

It is this set of problems to which we direct our attention in this thesis as we develop and present a convergent numerical approximation to minimum path problems. We intend to show a technique for finding such minimal paths which can be applied to arbitrary continuous manifolds in a general $\mathbb{R}^n$ space.

In the course of our investigation we propose and evaluate a method for obtaining numerical solutions to problems of geodesics and to problems of variational calculus in general.

2 HISTORICAL BACKGROUND

The mathematics of descriptive geometry and of the calculus of variations are fundamental to the questions of extremization under constraint. Their roots can be traced back to antiquity. Special credit must be given to the contributions of the 18th and 19th centuries, especially those of Euler, Lagrange, and those who followed.

2.1 Antiquity

Archimedes (287-212 BCE) defined the line as the shortest distance between two points lying in a plane [16]. In antiquity, the size of a city was commonly defined by its circumference. In this context, the topic of isoperimetrical shapes is mentioned by Zenodorus (circa 150 BCE) who stated without proof that the shape of largest enclosed area was the circle. Ptolemy (circa 150 CE) documents mapping ideas which we now refer to as the stereographic projection where he explains the concepts of projection and conformality.

2.2 Sixteenth and Seventeenth Centuries

Renaissance Europe saw a renewed interest in the mapping of the Earth. Mercator (1512-1594) introduced a projection in 1569 which maps meridians and parallels into straight lines. This "Mercator projection" stimulated a lively discussion in mathematics circles including later contributions from Leibniz.

Huygens (1629-1695) sought a means of measuring time exactly and looked for a pendulum motion that would have a period independent of the pendulum's altitude.

The solution to this problem, where a mass moves along a cycloid, was named the tautochrone.

Leibniz, between 1684 and 1692, published descriptions of the circle of osculation, and the significance of $d^2y = 0$ as a definition of an inflection point. What followed was a competition between Leibniz, Newton and the Bernoullis brothers which produced rapid advancements in the mathematics. Leibniz and Newton competed in calculus and hinted at variational ideas. It wasn't until Johann Bernoulli formulated the "brachistochrone curve" problem that interest in the subject of the calculus of variations gained momentum.

2.3   Eighteenth Century

Clairaut (1713-1765) investigated the analytic geometry of space as the intersection of surfaces. Euler (1707-1783) considered the topic of geodesics in a series of papers published between 1728 and 1732. He proved in 1736 that mass points move on a surface along geodesics in the absence of a force field [5]. His fundamental contribution to the topic of the calculus of variations, *Methodus inveniendi* [6], was published in 1744 [7]. He published a paper on the differential geometry of space curves in 1782.

Lagrange (1736-1813) further developed a systematic formulation of the ideas of Euler publishing a paper on the calculus of variations in 1762. In it he described the equations for a minimal surface. Lagrange and Euler collaborated on the calculus of variations mainly through a series of letters.

Monge (1746-1818) deserves special mention for his leadership role as well as his contributions to descriptive geometry. He cofounded the École Polytechnique in 1794 which attracted such students as Cauchy, Carnot, Liouville, Poisson, Mallat, Mandelbrot, Poincaré, Fresnel, and Ampère.

2.4 Nineteenth Century

During the 19th century, the center of progress moved from France to Germany where Gauss (1777-1855) established a new approach to the properties of surfaces, one depending only on the linear element and not the embedding in a higher dimensional space.

Jacobi (1804-1851) contributed his own ideas to those of Gauss, establishing "the existence of the conjugate points on the geodesics passing through a point on a surface [15]." Liouville (1809-1882) noticed that conformal transformations in space consist of inversions, simililarity and congruency transformations.

Riemann (1826-1866) presented ideas in an 1854 lecture at which Gauss was in attendance. In his lecture, Riemann introduced the concept (published later in 1867) of an n-dimensional manifold where both Euclidean geometry and non-Euclidean geometry are represented as special cases. The ideas of Riemann were continued and formalized by most notably by Grassmann (1809-1877), Lipschitz (1832-1903), Helmholtz (1821-1894), Lie (1842-1899), Poincaré (1854-1912).

3 THEORETICAL BACKGROUND

3.1 Calculus of Variations

3.1.1 Preliminaries

The calculus of variations is a field of mathematical analysis that deals with maximizing or minimizing functionals, which are mappings from a set of functions to the real numbers. Calculus of variations is considered to have begun with the contributions [17] of Johann Bernoulli who posed the brachistochrone problem which was quickly taken up by Jacob Bernoulli (brother of Johann), Euler, Lagrange, Legendre, Jacobi, Weierstrass and Hilbert. The subject is inextricably associated with classical mechanics [1, 2, 9, 12, 13].

We consider a normed linear function space H consisting of all continuous functions $y(x) : [a, b] \to \mathbb{R}$ with continuous first derivatives and with a norm defined as

$$\|y\|_1 = \max \|y(x)\| + \max \|y'(x)\|$$

and for the purposes of this section we call this set of such functions *admissible functions*.[1]

---

[1]The term admissible function, as it is commonly used in the literature, is in some sense relative to the specific variational problem we are addressing [8] in that it is a definition of the preconditions which must be met by a set of functions. In Section 4.2 we see that the CVA algorithm requires only boundedness and continuity, while the classical derivations in this section also require continuity in the first derivative.

Definition 3.1. [8] *A functional is a correspondence which assigns a real number to each function (or curve) belonging to some class.*

Definition 3.2. *The functional $J[y]$ for $y \in H$, is said to be continuous at the point $\hat{y} \in H$ if for all $\varepsilon > 0$, there exists $\delta > 0$ such that*

$$\|y - \hat{y}\| < \delta \implies |[J[y] - J[\hat{y}]| < \varepsilon.$$

Definition 3.3. *The functional $J[y]$ for $y \in H$, is said to be a linear functional if*

1. *$J[\alpha y] = \alpha J[y]$ for all $y \in H$ and $\alpha \in \mathbb{R}$,*

2. *$J[y_1 + y_2] = J[y_1] + J[y_2]$ for all $y_1, y_2 \in H$.*

Definition 3.4. *Let $J[y]$ be a functional with $y \in H$, and consider an increment of $\Delta J[y; h] = J[y + h] - J[y]$ for $h \in H$. Now separating the increment into a principle linear part and a residue, let*

$$\Delta J[y; h] = \delta J[y; h] + \varepsilon[y; h] \|h\|.$$

If

$$\lim_{\|h\| \to 0} \varepsilon[y; h] = 0$$

then we call $\delta J[y; h]$, or simply $\delta J[y]$, the differential (or variation) of $J[y]$, and $J[y]$ is said to be differentiable.

Definition 3.5. *Given* $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ *then*

$$\mathbf{J} = \frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}$$

is defined as the Jacobian matrix of $\mathbf{f}$*, and furthermore if* $m = n$*, then its determinant is called the Jacobian determinant.*

Theorem 3.6. *If a differentiable functional* $J[y]$ *has an extremum at* $y = \hat{y}$ *then*

$$\delta J[y; h] = 0$$

for $y = \hat{y}$ *and all admissible* h*.*

3.1.2 Classical Methods

We can separate calculus of variations solution techniques into two broad categories, classical (analytical) methods and direct (numerical) methods.

Euler noticed that a Taylor series expansion of a functional increment conveniently maps into the form of a principle part and a residue, with the residue containing vanishing second order and higher terms. By equating this principle part to zero (a property of an extremum) he obtained a differential equation which we now refer to as Euler's equation. Euler's equation can be viewed as a transform from a variational expression into a differential equation, thus opening the possibility of an analytical solution to problems of extremals.

Theorem 3.7 (Euler's Equation [8])**.** *Let $J[y]$ be a functional of form*

$$J[y] = \int_a^b F(x, y, y')dx,$$

defined on the set of functions $y(x)$ which have continuous first derivatives in $[a, b]$ and satisfy the boundary conditions $y(a) = A$, $y(b) = B$. Then a necessary condition for $J[y]$ to have an extremum for a given function $y(x)$ is that $y(x)$ satisfy Euler's equation

$$F_y - \frac{d}{dx}F_{y'} = 0. \tag{3.1}$$

3.1.2.1   Euler's equation in parameterized form

Implicit in the representation $y = y(x)$ is the requirement that $y(x)$ be a single-valued function of x. We can free ourselves of this restriction by considering parameterized representations of the curve leading to the form

$$J[y] = \int_{t_0}^{t_1} F[t, x(t), y(t), y'(t)]dt$$

$$= \int_{t_0}^{t_1} G[t, x(t), y(t), \dot{x}(t), \dot{y}(t)]dt$$

where

$$x = x(t) : [t_1, t_2] \to [a, b]$$

$$y = y(t) : [t_1, t_2] \to [A, B]$$

$$\dot{x} = \frac{dx}{dt}$$

19

$$y' = \frac{dy}{dx}$$

$$\dot{y} = \frac{dy}{dt} = \frac{dy}{dx}\frac{dx}{dt} = y'\dot{x}$$

as demonstrated in [18]. In this parameterized form, the associated Euler's equation becomes

$$G_x - \frac{d}{dt}G_{\dot{x}} = 0, \quad G_y - \frac{d}{dt}G_{\dot{y}} = 0. \tag{3.2}$$

See [18] for details.

We next demonstrate the application of Euler's equation to some problems of historical importance.

3.1.2.2   Example: Geodesic on the plane

Consider a curve represented by $y(x) : [a, b] \to \mathbb{R}$ where $y(a) = A$ and $y(b) = B$. We wish to find the curve $y(x)$ with the shortest path length (the geodesic on the plane). We restate this problem in terms of its functional representation.

$$J[y] = \int_a^b F(x, y, y')dx$$

$$= \int_a^b \sqrt{1 + (y')^2}dx$$

so

$$F(x, y, y') = (1 + (y')^2)^{1/2}$$

We use Euler's equation (3.1) to transform this problem into its corresponding differential equation

$$F_y - \frac{d}{dx}F_{y'} = 0$$

where

$$F_y = 0$$

$$F_{y'} = \frac{y'}{(1+(y')^2)^{1/2}}$$

$$\frac{d}{dx}F_{y'} = 0$$

$$F_{y'} = \frac{y'}{(1+(y')^2)^{1/2}} = constant$$

and after some basic algebraic manipulation we have the differential equation

$$y' = constant$$

with the well-known solution

$$y(x) = C_1 x + C_2$$

which is the equation of a straight line. After applying the boundary conditions we reach the final form of

$$y(x) = \left(\frac{B-A}{b-a}\right)x + \frac{Ab - Ba}{b-a}$$

as the equation of the shortest path between two points (a, A) and (b, B) on the plane.

3.1.2.3 Example: Brachistochrone

Perhaps a more interesting (less obvious) problem is the one posed by John Bernoulli where we are asked to find the curve which minimizes the transit time of a heavy particle starting at rest and moving under the force of gravity along a curve in the vertical plane. We note that the velocity of an object falling in a uniform gravitational field, according to Newtonian physics, is $v = \sqrt{2gy}$, and that the transit time is the integral of velocity over the path.

We can restate this problem in terms of its functional representation, choosing a convenient frame of reference with $(0,0)$ as the origin and y increasing during the transit, and terminating with the condition $y'(b) = 0$ corresponding to the condition reached at the "bottom" of the curve. The corresponding functional is

$$T[y] = \int_0^b F(x, y, y')dx$$

$$= \int_0^b \frac{\sqrt{1 + (y')^2}}{\sqrt{2gy}}dx$$

so

$$F(x, y, y') = \frac{\sqrt{1 + y'^2}}{\sqrt{2gy}} = \left(\frac{1 + y'^2}{2gy}\right)^{1/2} \tag{3.3}$$

Now we use Euler's equation (3.1) to transform this problem into its corresponding differential equation. We notice that F is not dependent on x so we can proceed from Euler's equation

$$F_y - \frac{d}{dx}F_{y'} = 0$$

to obtain

$$\left[F_y - \frac{d}{dx}F_{y'}\right]y' = 0$$

$$F_y y' - y'\frac{dF_{y'}}{dx} = 0$$

$$\frac{dF}{dx} - F_{y'}y'' - y'\frac{dF_{y'}}{dx} = 0$$

$$\frac{d}{dx}[F - y'F_{y'}] = 0$$

and consequently

$$F - y'F_{y'} = C, \text{ a constant}$$

but also

$$y'F_{y'} = \frac{(y')^2}{\sqrt{y(1 + (y')^2}}$$

which leads to

$$\sqrt{\frac{1 + (y')^2}{y}} - \frac{(y')^2}{\sqrt{y(1 + (y')^2)}} = C$$

$$y(1 + (y')^2) = \frac{1}{C^2} = A$$

and finally

$$x = \int \sqrt{\frac{y}{A - y}}\,dy$$

By performing the substitution

$$y = \frac{A}{2}(1 - \cos\theta) = A\sin^2(\theta/2) \tag{3.4}$$

it follows that

$$x = \int \sqrt{\frac{\sin^2(\theta/2)}{1 - \sin^2(\theta/2)}} A \sin(\theta/2) \cos(\theta/2) \mathrm{d}\theta$$

$$x = \int \sin^2(\theta/2) \mathrm{d}\theta$$

$$x = \frac{A}{2}(\theta - \sin\theta) + B.$$

The constant B becomes zero if we set $\theta = 0$ and $y = 0$ as our initial conditions, and by recalling (3.4) we then have the solution

$$x = \frac{A}{2}(\theta - \sin\theta)$$

$$y = \frac{A}{2}(1 - \cos\theta)$$

which is the general solution of a cycloid, due to Euler [6] for $\theta \in [0, \pi]$.

3.1.3 Direct Methods

Many practical variational problems are difficult to solve with analytical methods, and for these problems we turn to computational or direct methods. The direct methods attempt to find a minimizing sequence which converges on a solution. Such methods trace their origin back to Euler himself who proposed a direct method which we examine in this section, followed by a method due to Ritz [14].

3.1.3.1 Euler's method

Consider the extremum of the functional

$$J[y] = \int_{x_0}^{x_n} F(x, y, y')dx$$

where

$$y(x_0) = y_0 y(x_1) = y_1$$

Euler proceeded by partitioning the interval $[x_0, x_n]$ into n equal parts each of length

$$h = \frac{x_n - x_0}{n}$$

The functional may then be approximated by the following series

$$J[y_i] = h \sum_{i=1}^{n-1} F(x_0 + ih, y_i, \frac{y_{i+1} - y_i}{h})dx$$

with extrema of y_i such that

$$\frac{\partial J}{\partial y_i} = 0$$

The resulting system of $n - 1$ linear equations transforms a variational problem into a problem of linear algebra.

3.1.3.2 Ritz method

Consider the extremum of the functional

$$J[y] = \int_{x_0}^{x_1} F(x, y, y')dx$$

where

$$y(x_0) = y_0 \, y(x_1) = y_1$$

Ritz proceeded by approximating the functional with a linear combination of basis functions each of which also satisfy the boundary conditions

$$\overline{y}(x) = \alpha_0 b_0(x) + \alpha_1 b_1(x) + \cdots + \alpha_n b_n(x)$$

$$J[\overline{y}] = \int_{x_0}^{x_1} F(x, \overline{y}, \overline{y}')dx$$

where the extremum requires

$$\frac{\partial J[\overline{y}]}{\partial \alpha_i} = 0, \quad i = 0, 1, \ldots, n.$$

Finite element or spline-based basis functions are commonly used for Ritz method approximations.

4   A NUMERICAL APPROACH TO THE CALCULUS OF VARIATIONS

4.1   An overview of the CVA algorithm

It is our intention to describe a numerical method which can be applied to find solutions for minimal path problems. We describe in this section what we call the "CVA algorithm" (CVA, pronounced "see-va"), a numerical approximation to the solution of problems in variational calculus.

We take some inspiration from Cooley and Tukey [4] in their elegant formulation of the fast fourier transform, in that we approach our task by breaking our problem down to its smallest element. We solve that element and then use it to build up a full solution from these basic parts. We call this basic element the "minpoint approximation" and describe it in Section 4.1.2. Next, we describe in Section 4.1.3 a numerical method which will build minimal piecewise paths from minpoints. We call this technique "straddling." Finally, we describe in Section 4.1.4 a third numerical method which we call "refinement." Refinement produces paths which are arbitrarily close to minimum path integrals.

While the mathematics is general (see Section 4.2), our descriptions in this section are intended to be heuristic, explanatory, and accessible. For that reason we begin with the simple case of a 2-dimensional (u, v) parameterized surface embedded into a 3-dimensional space following the conventions established by the mathematics of differential geometry [11]. Our first example is that of the geodesic on the unit sphere. The solution of the geodesic on a unit sphere makes a fine example since it is a non-trivial problem and one for which an analytical solution is well known.

27

4.1.1 Models and Metrics

We need a framework for our work. For that we create a 2-dimensional $\langle u, v \rangle$ unit plane and a mapping G between points in that plane and points on a corresponding 3-dimensional unit sphere.[2] Think of u as a "latitude" and v as a "longitude." We use the conventions $u = 0$ to represent the "north pole", $u = 1$ the "south pole", and $v \in [0, 1]$ where $v = 0.5$ is on the "prime meridian." So far we have described a model as in Figure 1 where we show a point in the $\langle u, v \rangle$ parameter space and its corresponding point on the surface of the sphere.
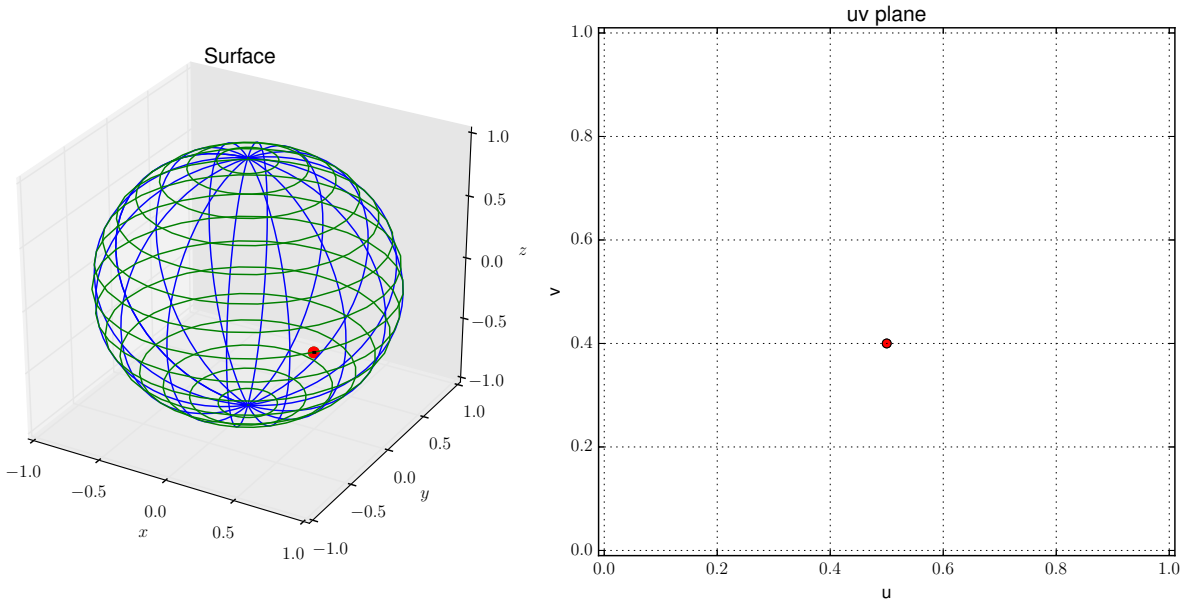
Figure 1: Model – Unit Sphere

---

[2]The CVA algorithm is applicable to the general case of m-dimensional parameter spaces mapping into n-dimensional hypersurfaces. See Section 4.5 for example solutions in higher dimensional spaces.

To measure distances between points in this space we need a metric. We use the Euclidean distance metric[3] in our first example. Here we depict a direct path through the center of the sphere. The metric provides its length.
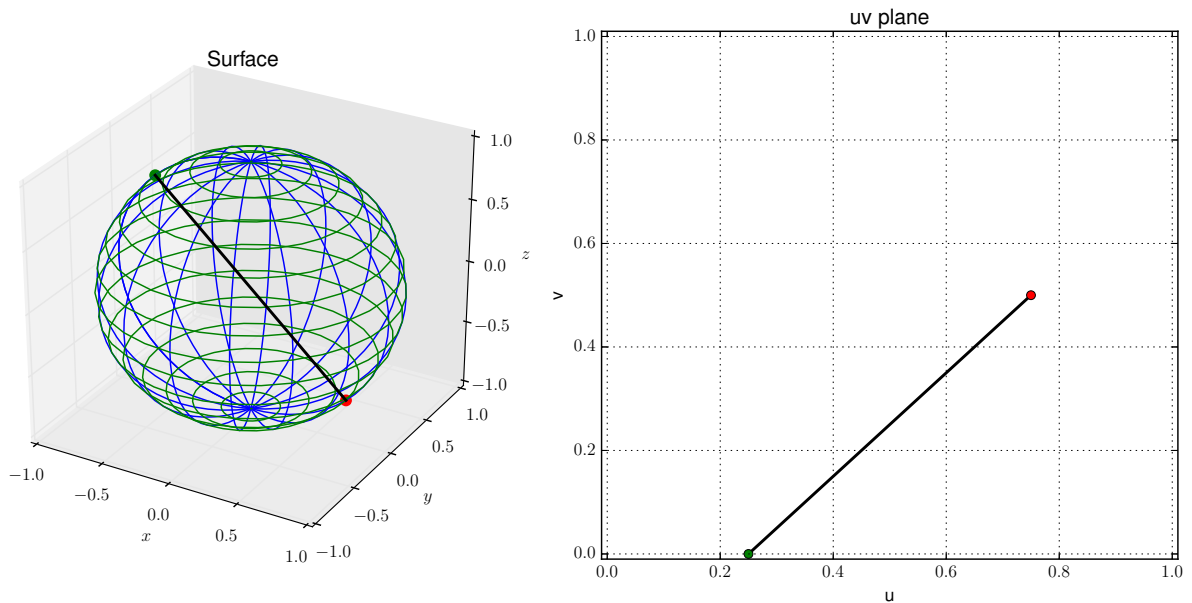
Figure 2: Euclidean Distance Metric

Using a framework equipped with a suitable model and an associated metric, we are prepared in the next section to turn our attention to the task of finding minimum paths.

---

[3]The CVA algorithm is valid for the general case of any bounded and continuous metric. See Section 4.5 for example solutions using other metrics such as brachistochrone, Schwarzschild, and Minkowski.

4.1.2 Minpoint approximation

In order to find minimum paths we break our problem down to its smallest element. Consider that we want to find the minimum path across the surface of a sphere from one point, call it $\mathbf{u}_a$ (shown as a green dot) to another, call it $\mathbf{u}_b$ (shown as a red dot). These two points map onto the surface of the sphere as points that we will call $\mathbf{x}_a$ (also green) and $\mathbf{x}_b$ (also red).
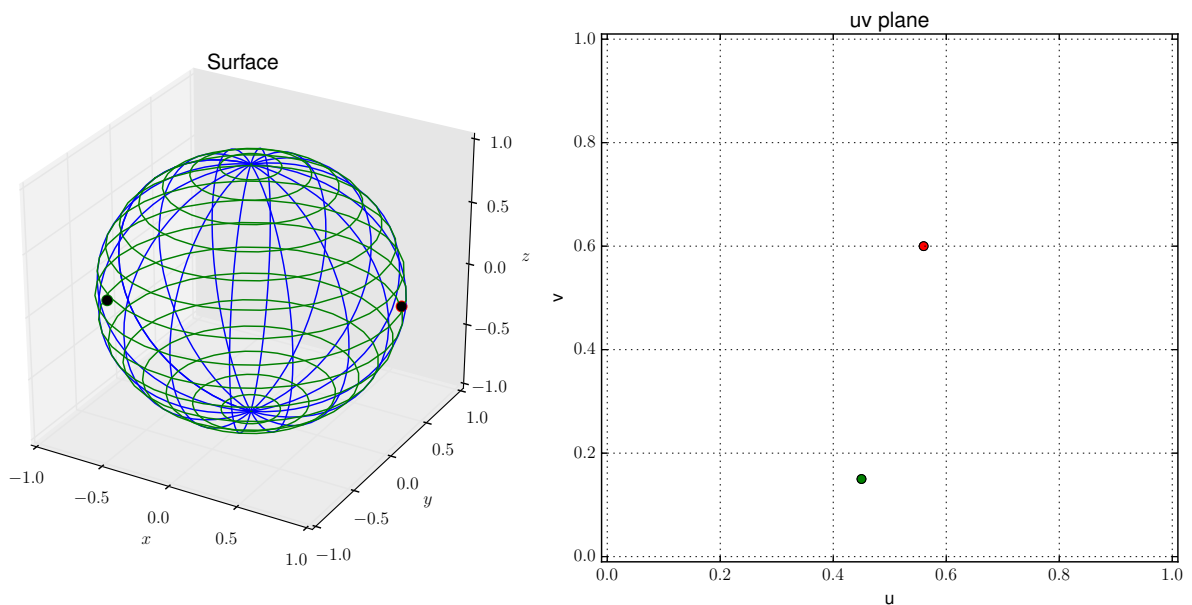
Figure 3: Starting and ending points
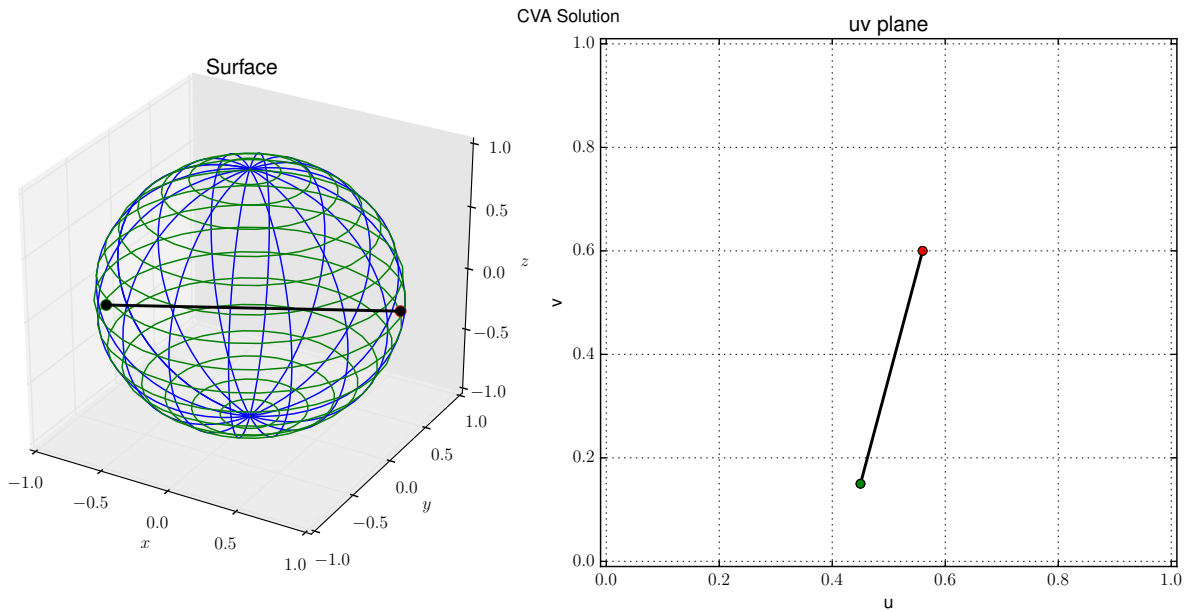
Figure 4: Secant vector from $\mathbf{u}_a$ to $\mathbf{u}_b$

We first draw a line between $\mathbf{u}_a$ and $\mathbf{u}_b$ and we call this our secant vector. The secant vector is a first approximation of our minimum path. We can improve our accuracy by introducing a third point into our solution which we will call the "minpoint." Let's call it $\mathbf{u}_{min}$ with its corresponding surface point $\mathbf{x}_{min}$. We require that a minpoint be found such that the length of the path from $\mathbf{x}_a$ to $\mathbf{x}_{min}$ and then $\mathbf{x}_{min}$ to $\mathbf{x}_b$ be the minimum of all possible trial paths. For that we use an iterative solution.

First, we find the point in the $\langle u, v \rangle$ plane at the center between $\mathbf{u}_a$ and $\mathbf{u}_b$ which is $\mathbf{u}_m = (\mathbf{u}_b - \mathbf{u}_a)/2$. Then we find a perpendicular to the segment through that point. We call that perpendicular a "trial space" and we locate a set of points on it within an initial *radius* of $\mathbf{u}_m$. Our model with its initial set of trial points is shown in Figure 5.
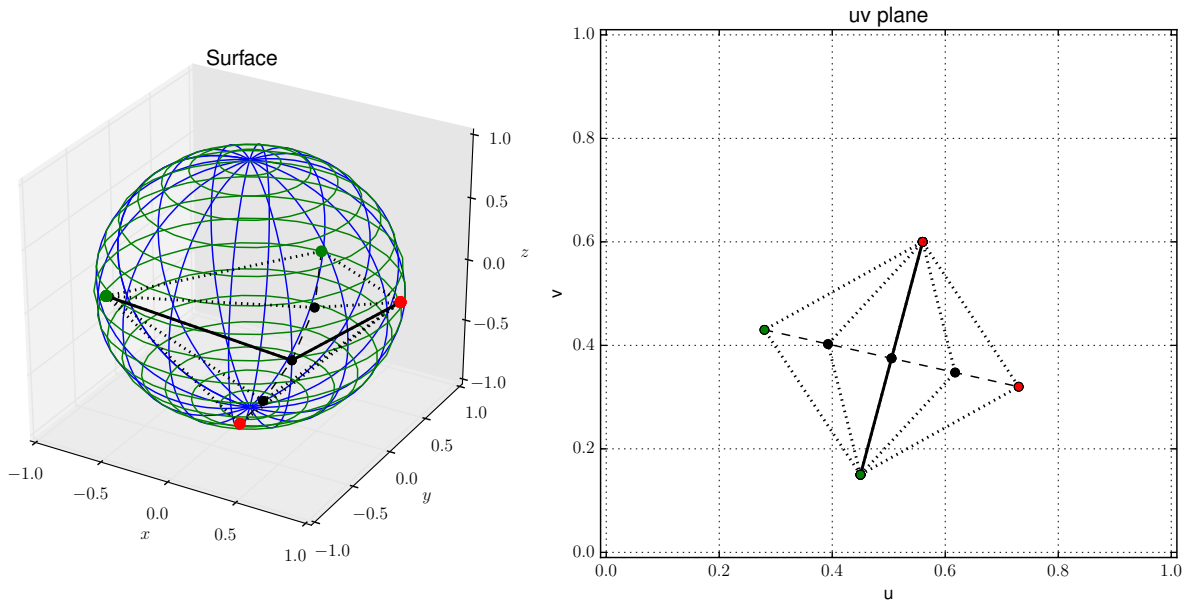
Figure 5: Initial trial space

We use our metric to measure each of the five possible paths connecting $\mathbf{x}_a$ and $\mathbf{x}_b$. Recalling the extreme value theorem, if we find a point surrounded by two points of higher value then we know that a minimum lies between the two neighboring points. We select that minimum path and let it define our new $\mathbf{u}_m$. At the same time we can now restrict our *radius* of interest to $1/2$ of its previous value. Repeating the procedure gives us ever improving approximations
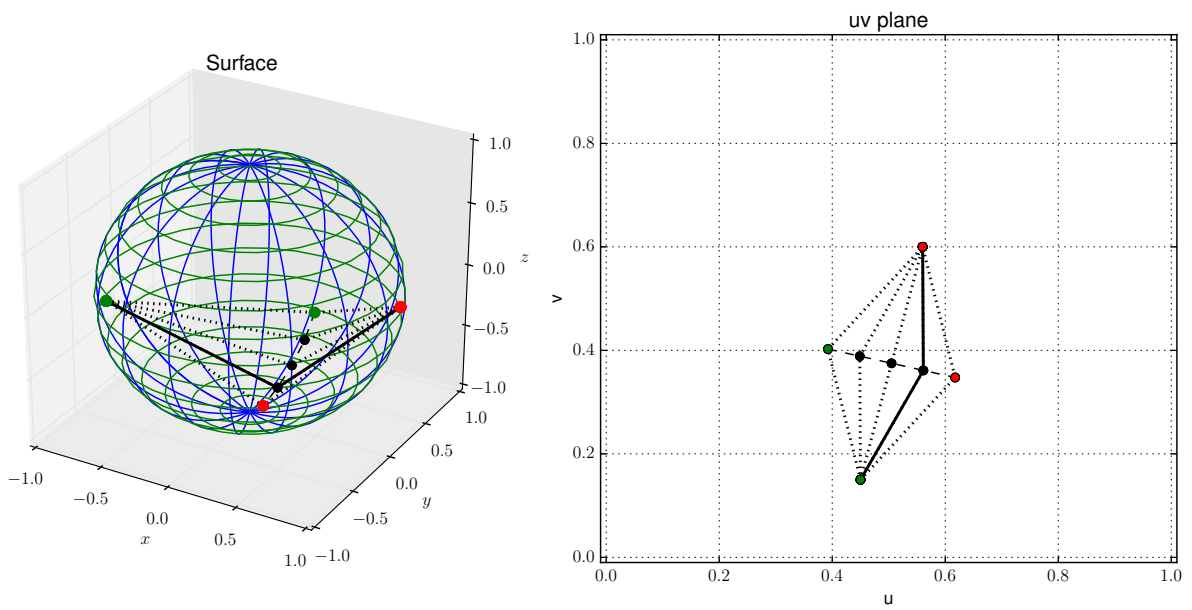
Figure 6: Trial space after iteration on Figure 5
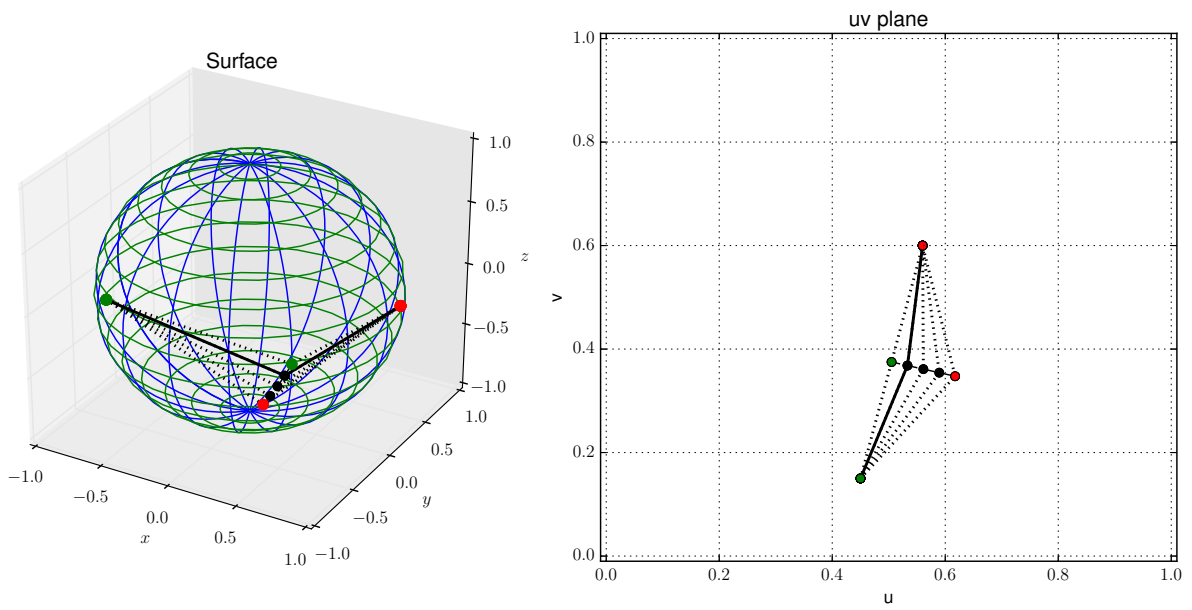
Figure 7: Trial space after iteration on Figure 6

33
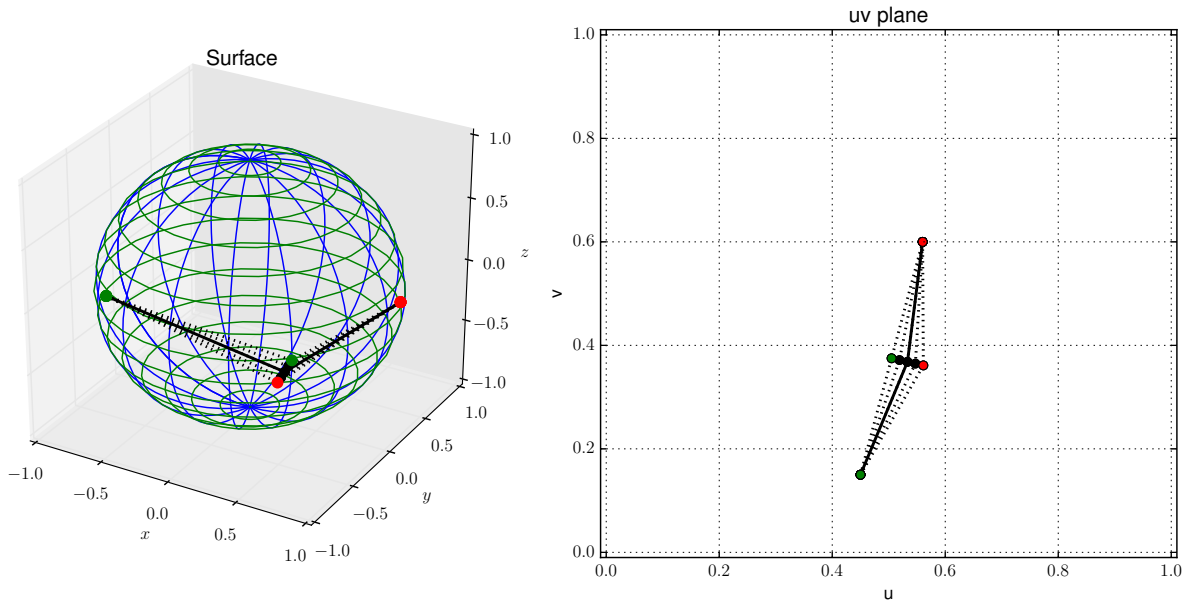
Figure 8: Trial space after iteration on Figure 7

By continuing this process, we reduce our error in each step until we have approx-

imated the optimal location of $\mathbf{u}_{min}$ to whatever accuracy we desire.
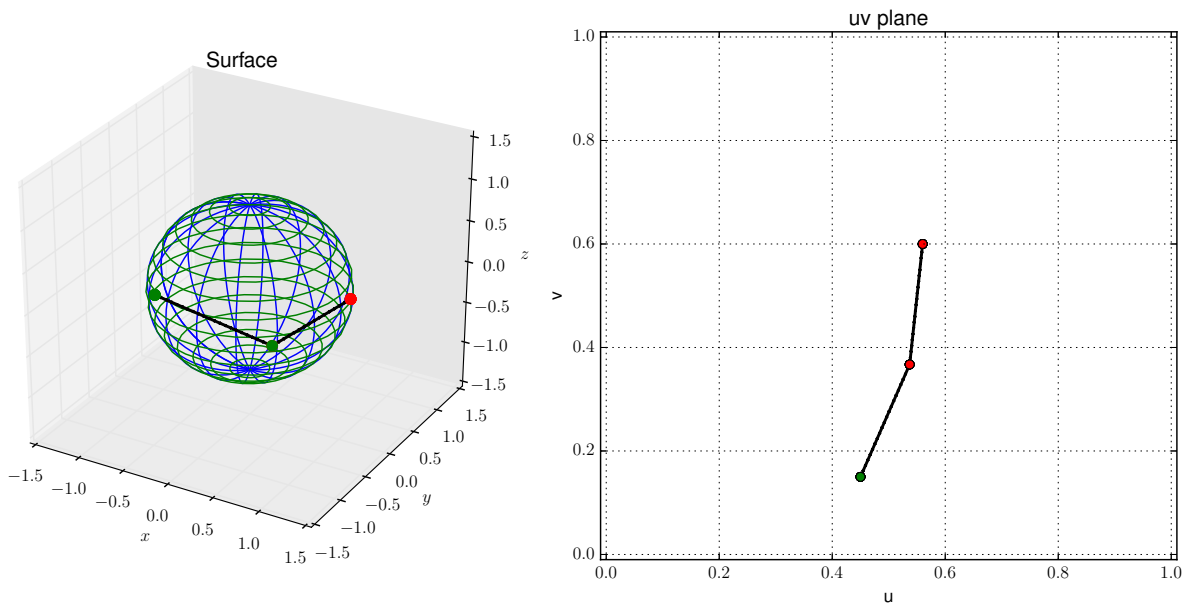
Figure 9: $\mathbf{u}_{min}$ after n iterations

With this procedure we have found the point on our trial space that defines a minimal path of three points.
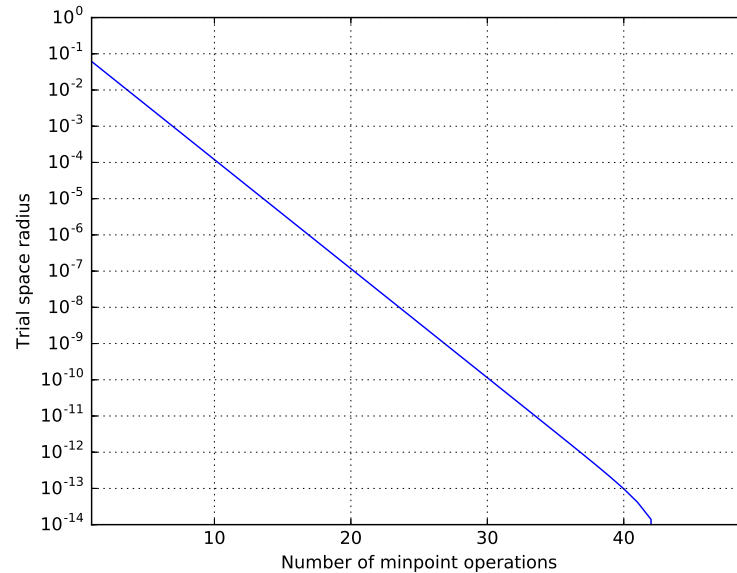
Figure 10: Minpoint convergence

We can see in Figure 10 that we can continue to apply minpoint operations until we have located our result inside a radius as small as we desire.

In this section we described the minpoint approximation which is the basic computational element on which the CVA algorithm is based. We will show in the following sections how we can use this basic element to build minimal paths with larger numbers of points.

4.1.3 Straddling

In the previous section we saw how we could begin with knowledge of two endpoints and then proceed to construct an intermediate point which we called the minpoint. Now let's consider the problem of building a longer sequence which connects our two endpoints.

Continuing with the same model and metric as in Section 4.1.2, let's move from a three-point path to a five-point path by adding an additional point between each of our previous points[4]. Let's call these five points $(\mathbf{u}_0, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3, \mathbf{u}_4)$, where $\mathbf{u}_0 = \mathbf{u}_a$, our original starting point, $\mathbf{u}_4 = \mathbf{u}_b$, our original ending point, and $\mathbf{u}_2 = \mathbf{u}_{min}(\mathbf{u}_a, \mathbf{u}_b)$, the minpoint between $\mathbf{u}_a$ and $\mathbf{u}_b$.

We now have a five-point piecewise curve connecting $\mathbf{u}_a$ and $\mathbf{u}_b$. As a first step we can use our minpoint approximation to set initial values for these new points based on the values of the neighbors. This gives us the path in Figure 11.

---

[4]Although we have chosen a path of length five as our example in this section, we should note that the straddling operation is valid for sequences of arbitrary length.
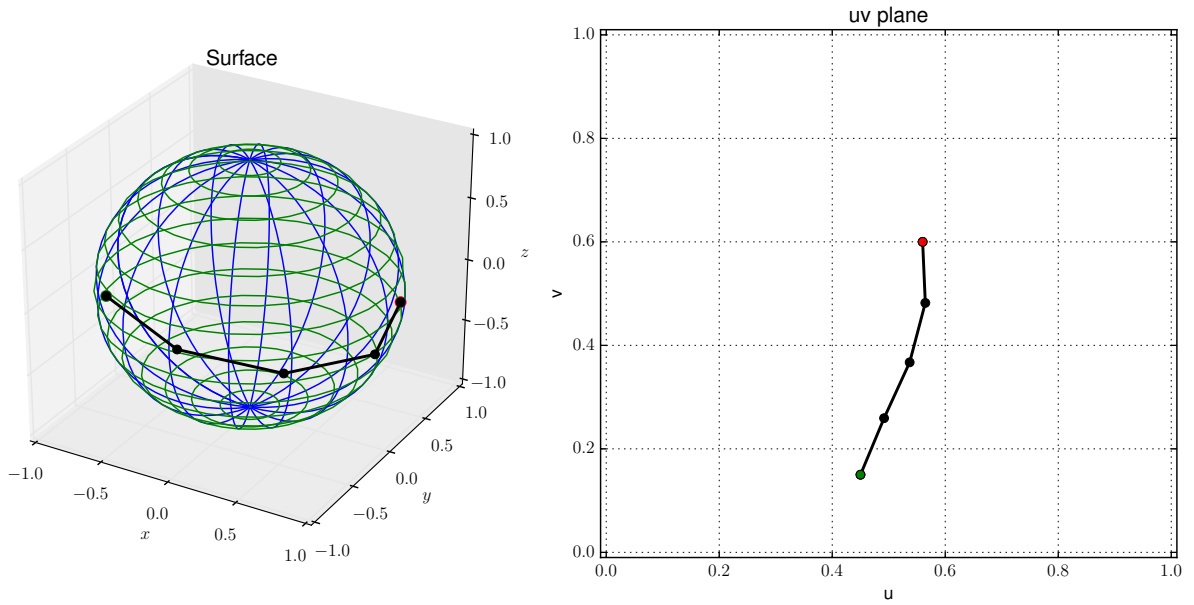
Figure 11: Five-point path after an odd straddle

By producing updated values for each odd numbered point based on neighbors we have moved our curve in the direction of the minimal. However, moving points with an odd index means points with even indices are no longer minpoints. In the next step we adjust each interior point with an even index (in our example that would be $\mathbf{u}_2$) so that it is the minpoint of its new neighbors. This gives us the path in Figure 12.
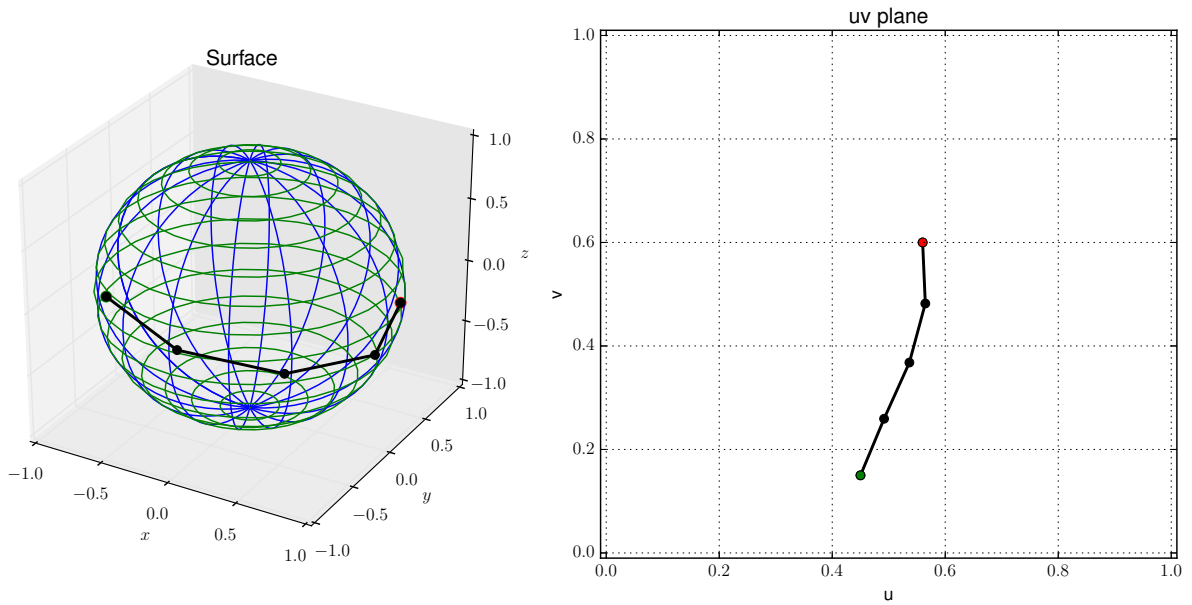
Figure 12: Five-point path after an even straddle

Continuing this procedure of alternating odd/even straddle operations reduces the length of the path on the sphere at each step until we have approximated a minimal piecewise path to whatever accuracy we desire.

In order to see this convergence better, Figure 13 illustrates on a logarithmic scale how the total path summation varies as the number of straddle operations increases.
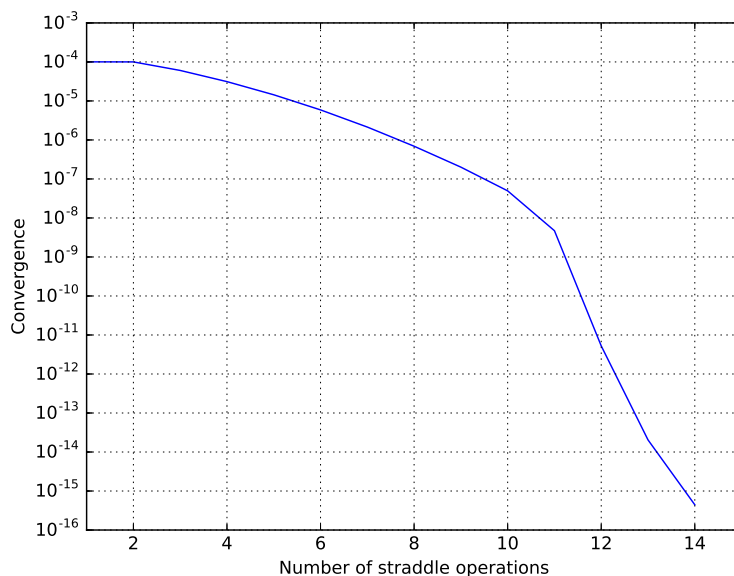
Figure 13: Straddle convergence

Figure 13 indicates convergence to the limit of 64-bit floating point representation after fewer than 15 straddle operations.

At this point we have the machinery in place to approximate minimal paths for piecewise curves. What remains to be covered in the next section will be the refinement of this approach to form paths which lie arbitrarily close to our surface.

4.1.4 Refinement

In the previous section we saw how we could begin with knowledge of two endpoints and then proceed to construct minimal piecewise paths. What remains is to successively expand the number of points in our solution curve so as to get as close to a minimal path integral as we desire. To do that we refine our solution by introducing an increasing number of points in a succession of straddle-converged paths.

To illustrate the refinement process, we begin with only a knowledge of a starting point and an ending point, and after one refinement we have a three-point curve.
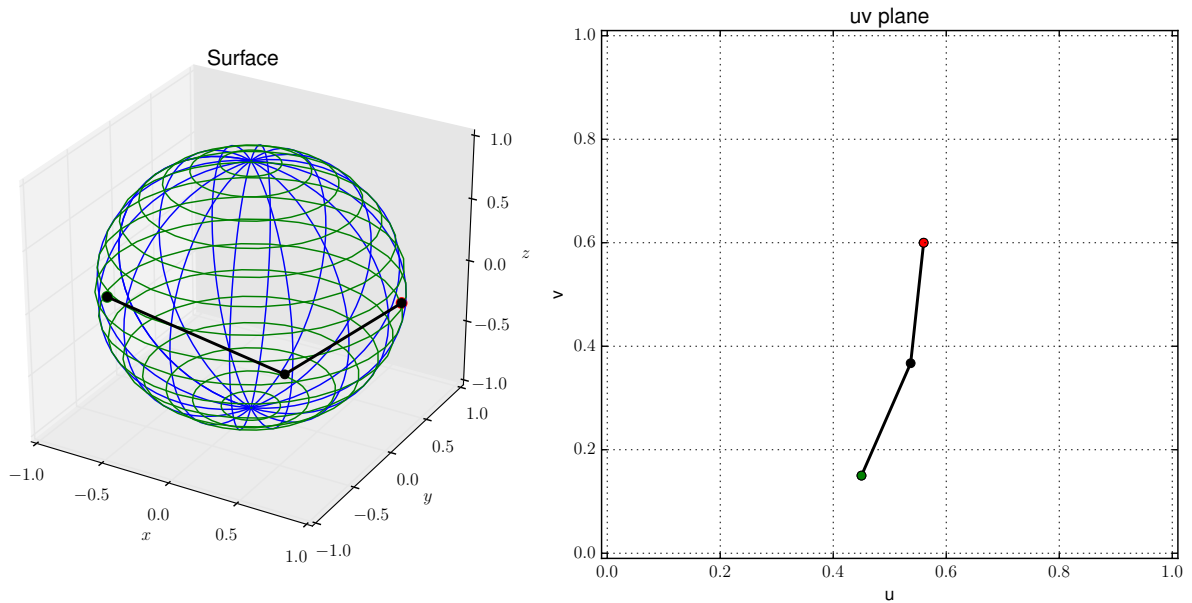
Figure 14: After first refinement

A second refinement adds points to our path and places us closer to a minimal path on the surface.
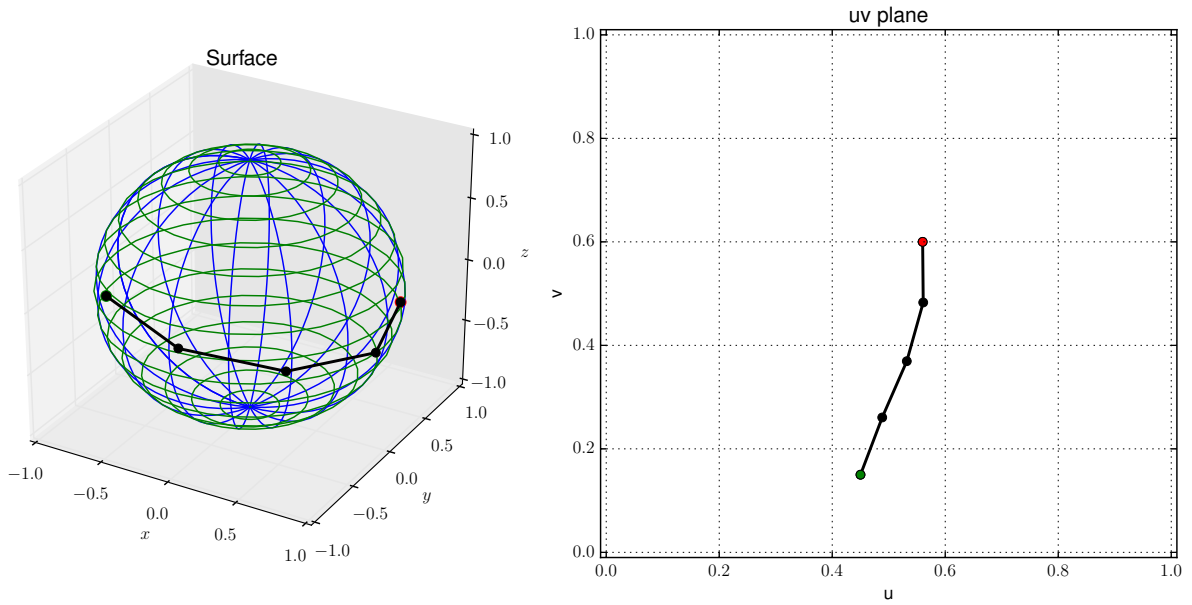
Figure 15: After second refinement

A third refinement is a reasonable approximation in this first example.
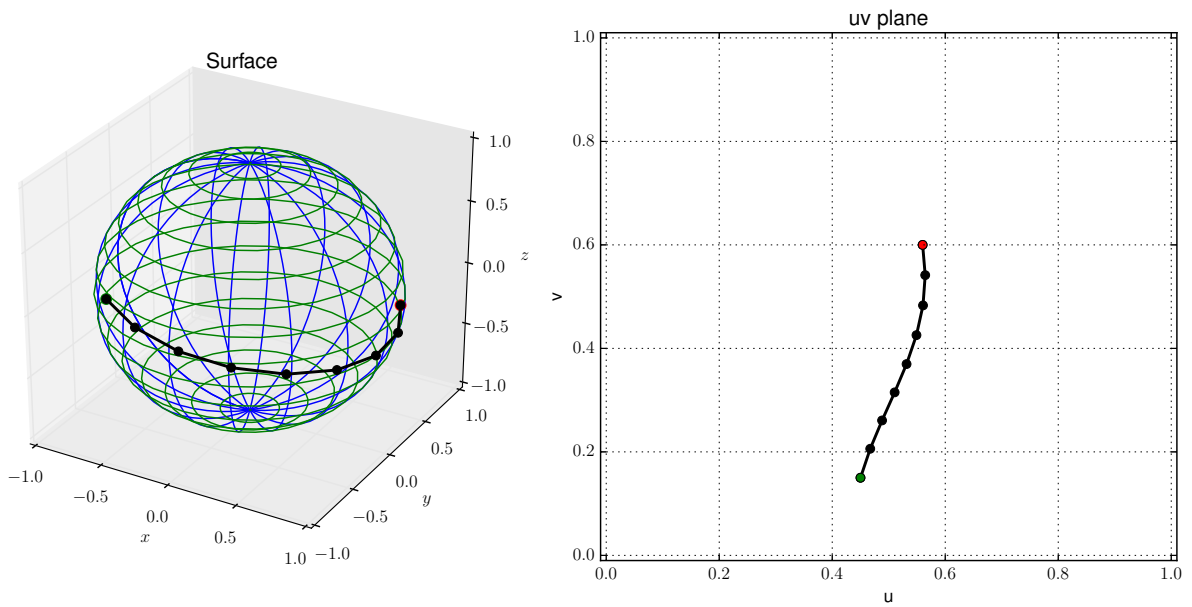
Figure 16: After third refinement
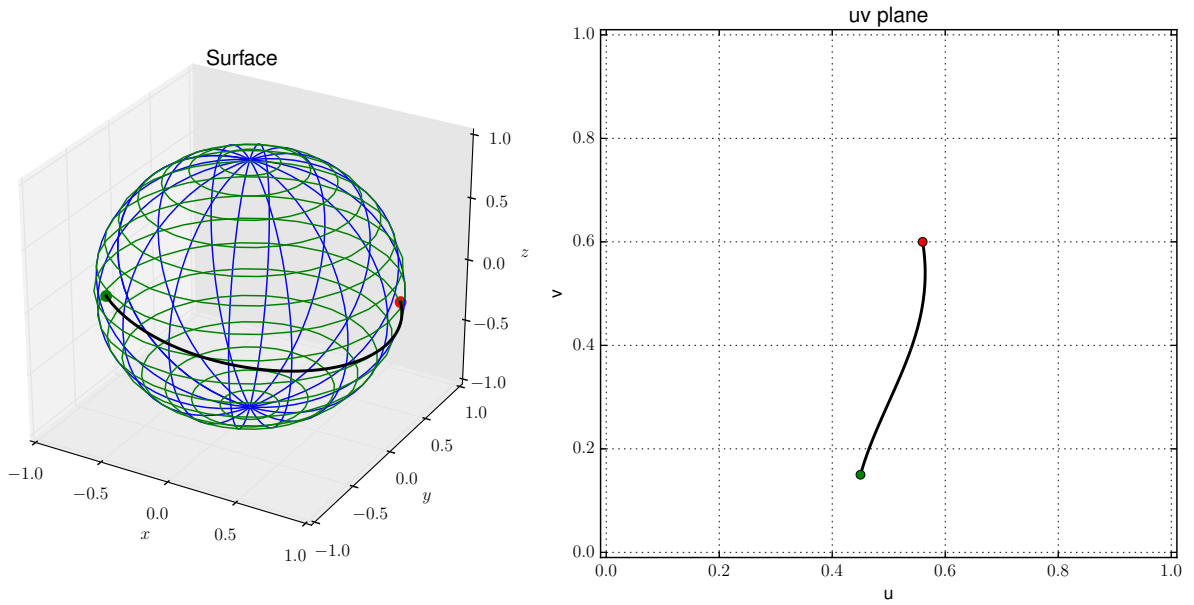
Figure 17: After n^{th} refinement

Continuing the process of refinement, we reduce our error in each step until we have approximated the geodesic curve between $\mathbf{u}_a$ and $\mathbf{u}_b$ to whatever accuracy we desire.

In Section 4.4 we look in more detail into the validation of these results, but before we leave this section we include Figure 18 which illustrates the convergence curve for the overall algorithm as a log plot.
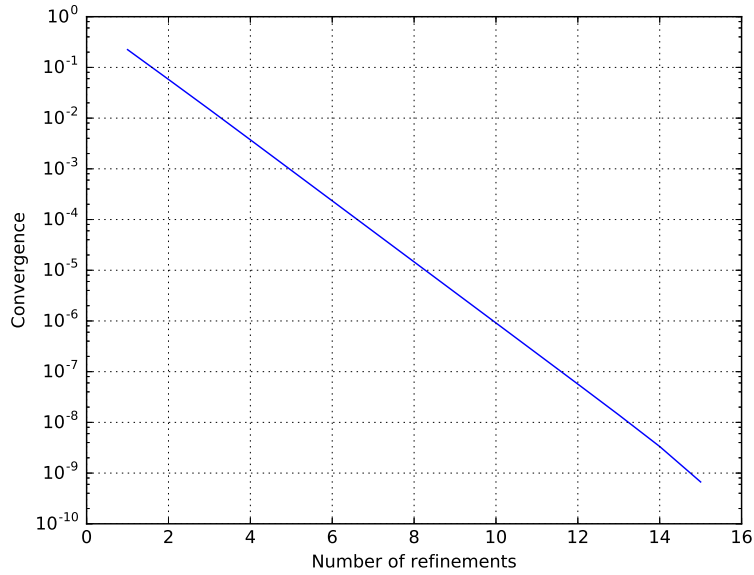
Figure 18: Refinement convergence

4.2   The CVA algorithm described through mathematics

It is our intent in this section to provide a formal proof of the validity of the CVA algorithm. Our approach to this task is taken in four parts. We first establish the mathematical context in which we work, then we define and prove the validity of each component of the algorithm beginning with the minpoint concept, then the straddling operation, and finally the refinement phase. We show that a given set of preconditions implies convergence to a minimal.

4.2.1   Context

Before we can mathematically justify the CVA algorithm, we must first rigorously define the context in which we develop the algorithm itself. We begin with a general

m-manifold representation from which we take a single chart as our mapping from a parameter space to a hypersurface.

Definition 4.1. *Given an m-manifold Ω with a collection of continuous 1-1 mappings $G : \mathbb{U}^m \to \Omega$ where $\mathbb{U}^m$ is a bounded open subset of $\mathbb{R}^m$, then we call each G a chart and we call the collection of charts an atlas if for any pair G_1, G_2 of overlapping charts, there exists a $1 - 1$ map ϕ such that $G_1(\mathbf{u}) = G_2(\phi(\mathbf{u}))$ for all $\mathbf{u}$ in the overlap.*

Definition 4.2. *Given $\mathbf{u} = \langle u_0, u_1, \ldots, u_{m-1} \rangle$, $\mathbf{u} \in \mathbb{U}^m$ with some chart $G \in \Omega$, then we call $\mathbf{u}$ a parameterization point, and we call the set of all such points, $\mathbb{U}^m$ an m-dimensional parameter space.*

Definition 4.3. *Given a chart $G : \mathbb{U} \to \Omega$ then we call the set of points $\mathbb{X} \subset \Omega$, $\mathbb{X} = \{\mathbf{x} \mid \mathbf{x} = G(\mathbf{u}) \, \forall \, \mathbf{u} \in \mathbb{U}\}$ a hypersurface.*

Definition 4.4. *Given a function $\alpha : [a,b] \to \mathbb{U}$ for $a, b \in \mathbb{R}$, then the set of points $S_u = \{\mathbf{u} \mid \mathbf{u} = \alpha(t), t \in [a,b], \mathbf{u} \in \mathbb{U}\}$ is called a parameterization curve. The set of points $S_s = \{\mathbf{x} \mid \mathbf{x} = G(\mathbf{u}), \, \mathbf{u} \in S_u\}$ is called a hypersurface curve or a path.*

Definition 4.5. *A metric is a continuous function $M : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ that assigns a real number to any pair of points $\mathbf{x}, \mathbf{y} \in \mathbb{X}$ and satisfies $M(\mathbf{x}, \mathbf{y}) = 0$ whenever $\mathbf{x} = \mathbf{y}$, $0 \leq M(\mathbf{x}, \mathbf{y}) < \infty$ for all $\mathbf{x}, \mathbf{y}$ and satisfies $M(\mathbf{a}, \mathbf{c}) \leq M(\mathbf{a}, \mathbf{b}) + M(\mathbf{b}, \mathbf{c})$ for all $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{X}$.*

Definition 4.6. *Given a chart G and a pair of parameterization points $\mathbf{u_a}, \mathbf{u_b} \in \mathbb{U}^m$, and given $\mathbf{u_m} = \frac{\mathbf{u_b} - \mathbf{u_a}}{2}$, and given $\{\mathbf{u_\perp}\}$ to be a set of $m-1$ orthonormal basis vectors*

perpendicular to $\mathbf{u_b} - \mathbf{u_a}$, *then the set of points which is spanned by* $\{\mathbf{u_\perp}\}$ *and which includes the parameterization point* $\mathbf{u_m}$ *is called the trial space* $\mathbb{T}$, *which we write* $\mathbb{T}(\mathbf{u_a}, \mathbf{u_b})$.

4.2.2   Minpoint

We now define the concept of a minpoint showing its relationship to the trial space. We introduce an operator which we use to sum paths in our context.

Definition 4.7. *Given a chart* G *and a pair of parameterization points* $\mathbf{u_a}, \mathbf{u_b} \in \mathbb{U}^m$ *with trial space* $\mathbb{T}$, *and given a metric* M, *and given*

$$J_{\mathbf{u_{min}}} = \min_{\mathbf{u_t} \in \mathbb{T}} \left[M(G(\mathbf{u_a}), G(\mathbf{u_t})) + M(G(\mathbf{u_t}), G(\mathbf{u_b})) \right]$$

then $J_{\mathbf{u_{min}}}$ *is called the minpoint functional and* $\mathbf{u_{min}} = \arg\min J_{\mathbf{u_{min}}}$ *is called the minpoint of* $\mathbf{u_a}, \mathbf{u_b}$ *which we write* $\mathbf{u_{min}}(\mathbf{u_a}, \mathbf{u_b})$.

Definition 4.8. *Given a chart* G *with metric* M *and a parameterization curve* S *consisting of* n *points, then the sum*

$$\mathcal{L}[S] = \sum_{i=0}^{n-2} M(G(\mathbf{u_i}, \mathbf{u_{i+1}}))$$

is called a piecewise path summation and $\mathcal{L}[S]$ *is called the piecewise path operator.*

4.2.3 Straddling

The straddling concept is defined in several steps. The straddle operator is used to generate a sequence of piecewise paths. The corresponding path summations are shown to be monotonically decreasing converging to a minimal piecewise path.

Definition 4.9. *Given a chart G with metric M, let $S_i \in \mathbb{U}$ be a parameterization curve containing $n + 1$ points, $S_i = (\mathbf{u}_{i,0}, \mathbf{u}_{i,1}, \ldots, \mathbf{u}_{i,n-1}, \mathbf{u}_{i,n})$. We then define an operator $\mathbb{S}$ such that*

$$S_{i+1} = \mathbb{S}[S_i]$$

where

$$S_{i+1} = (\mathbf{u}_{i+1,0}, \mathbf{u}_{i+1,1}, \ldots, \mathbf{u}_{i+1,n-1}, \mathbf{u}_{i+1,n})$$

and where if i is odd

$$\mathbf{u}_{i+1,j} = \begin{cases} \mathbf{u}_{i,0} & \textit{for } j = 0 \\[2ex] \mathbf{u}_{i,j} & \textit{for } j = even, j \neq 0, n \\[2ex] \mathbf{u}_{min}(\mathbf{u}_{i,j-1}, \mathbf{u}_{i,j+1}) & \textit{for } j = odd, j \neq 0, n \\[2ex] \mathbf{u}_{i,n} & \textit{for } j = n \end{cases}$$

and where if i is even

$$\mathbf{u}_{i+1,j} = \begin{cases} \mathbf{u}_{i,0} & \textit{for } j = 0 \\[2ex] \mathbf{u}_{min}(\mathbf{u}_{i,j-1}, \mathbf{u}_{i,j+1}) & \textit{for } j = even, j \neq 0, n \\[2ex] \mathbf{u}_{i,j} & \textit{for } j = odd, j \neq 0, n \\[2ex] \mathbf{u}_{i,n} & \textit{for } j = n \end{cases}$$

We call this operator a straddle operator which we write $\mathbb{S}[S_i]$.

Lemma 4.10. *Given a chart G with metric M and a parameterization curve S, then*

$$\mathcal{L}[\mathbb{S}[S]] \leq \mathcal{L}[S]$$

Proof. Given a chart G with metric M and a parameterization curve S_i and let $S_{i+1} = \mathbb{S}[S_i]$. Then consider any point $\mathbf{u}_{i,j} \in S_i$ such that i, j are either both odd or both even. For such points we have

$$\mathbf{u}_{i+1,j} = \mathbf{u}_{min}(\mathbf{u}_{i,j-1}, \mathbf{u}_{i,j+1})$$

The piecewise path summation between this point and its neighbors in S_i is

$$L_i = \mathcal{L}[\mathbf{u}_{i,j-1}, \mathbf{u}_{i,j}, \mathbf{u}_{i,j+1}]$$

The piecewise path summation between this point and its neighbors in S_{i+1} is

$$L_{i+1} = \mathcal{L}[\mathbf{u}_{i,j-1}, \mathbf{u}_{min}(\mathbf{u}_{i,j-1}, \mathbf{u}_{i,j+1}), \mathbf{u}_{i,j+1}]$$

It follows from the definition of the minpoint, Definition 4.7, that

$$L_{i+1} \leq L_i$$

and since by Definition 4.8 the piecewise path summation of S is the sum of the piecewise path summation of its subintervals, then

$$\mathcal{L}[S_{i+1}] \leq \mathcal{L}[S_i]$$

and consequently

$$\mathcal{L}[\mathbb{S}[S]] \leq \mathcal{L}[S] \quad \text{for all } S$$

$\square$

Definition 4.11. *Given a chart G with metric M and a parameterization curve $S_0 \in \mathbb{U}$ containing $n+1$ points, let $S_0 = (\mathbf{u}_{0,0}, \mathbf{u}_{0,1}, \ldots, \mathbf{u}_{o,n-1}, \mathbf{u}_{0,n})$. We then construct a sequence of parameterization curves as*

$$S_0 = (\mathbf{u}_{0,0}, \mathbf{u}_{0,1}, \mathbf{u}_{0,2}, \ldots, \mathbf{u}_{0,j}, \ldots \mathbf{u}_{0,n-1}, \mathbf{u}_{0,n})$$

$$S_1 = \mathbb{S}[S_0]$$

$$\vdots$$

$$S_i = \mathbb{S}[S_{i-1}]$$

We call this sequence a straddle sequence.

Theorem 4.12. *Given a chart G with metric M and with a straddle sequence S, then the piecewise path summation $\mathcal{L}[S]$ converges to a positive number.*

Proof. Given a chart G with metric M and let S denote a straddle sequence $\{S_n : n = (0, 1, 2...)\}$. From Definitions 4.5 and 4.1, and 4.8 it follows that $0 \leq \mathcal{L}[S_n] < \infty$. From Lemma 4.10 it follows that the straddle sequence is monotonically decreasing so we write

$$\infty > \mathcal{L}[S_0] \geq \mathcal{L}[S_1] \geq \cdots \geq \mathcal{L}[S_n] \geq \cdots \geq 0$$

Let $\varepsilon > 0$, and let $L = \min \mathcal{L}[S]$. Since $L + \varepsilon$ is not a lower bound for $\mathcal{L}[S]$, there exists N such that $\mathcal{L}[S_N] < L + \varepsilon$. Since $\mathcal{L}[S_n]$ is decreasing, we have $\mathcal{L}[S_N] \geq \mathcal{L}[S_n]$ for all $n \geq N$. Also $\mathcal{L}[S_n] \geq L$ for all n, so $n > N$ implies $L \leq \mathcal{L}[S_n] < L + \varepsilon$, and consequently

$$\lim_{n \to \infty} \mathcal{L}[S_n] = L.$$

$\square$

Definition 4.13. *Given a chart G with metric M and with a parameterization curve connecting two parameterization points $\mathbf{u}_a$ and $\mathbf{u}_b$, and let S denote the corresponding straddle sequence. If*

$$\mathcal{I}[\mathbf{u}_a, \mathbf{u}_b] = \lim_{n \to \infty} \mathcal{L}[S_n] = L$$

then we call L the straddle-converged piecewise path summation, and we call $\mathcal{I}[\mathbf{u}_a, \mathbf{u}_b]$ the minimal path operator.

Note: At this point in our development of the CVA algorithm we reach a key result, namely that straddle-converged piecewise path summations are minimal. Next, we proceed to state and prove this result.

Theorem 4.14. *Straddle-converged piecewise path summations are minimal.*

Proof. Given a chart G with metric M and with a parameterization curve connecting two parameterization points $\mathbf{u}_a$ and $\mathbf{u}_b$, and let S denote the corresponding straddle sequence where

$$\mathcal{I}[\mathbf{u}_a, \mathbf{u}_b] = \lim_{n \to \infty} \mathcal{L}[S_n] = L$$

and with S_L such that

$$L = \mathcal{L}[S_L]$$

Now suppose that there exists another parameterization curve S_α connecting $\mathbf{u}_a$ and $\mathbf{u}_b$ where $S_\alpha \neq S_L$ such that $\mathcal{L}[S_\alpha] < \mathcal{L}[S_L]$. From Theorem 4.12 we know that a straddle sequence is monotonically decreasing, and therefore if we take the limit of the straddle sequence, $\lim_{n \to \infty} \mathcal{L}[S_{\alpha_n}] = L_\alpha$, then $\mathcal{L}[S_\alpha] \geq L_\alpha = L_T$. This contradiction then

50

shows that $\mathcal{L}[S_L] \leq \mathcal{L}[S_\alpha]$ and thus the straddle-converged piecewise path summation, $\mathcal{L}[S_L]$, is minimal. $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\square$

4.2.4 Refinement

In problems of variational calculus we are concerned with functionals of the form

$$J[S] = \int_S M(G(s))\, ds$$

over curves S satisfying some given boundary conditions. Our focus is on minimizing such functionals over a set of parameterization curves connecting two parameterization points $\mathbf{u_a}$ and $\mathbf{u_b}$.

Definition 4.15. *Given a chart G with metric M, where $J = \min J[S]$ over a set of all parameterization curves connecting two parameterization points $\mathbf{u_a}$ and $\mathbf{u_b}$, then we call J a minimal path integral which we write $J[\mathbf{u_a}, \mathbf{u_b}]$. In the special case where M is a Euclidean distance metric we call an $\arg\min$ curve of J a geodesic of the hypersurface.*

Given a chart G with metric M and with a parameterization curve of length $2^n + 1$ connecting two parameterization points $\mathbf{u}_a$ and $\mathbf{u}_b$, and consider a sequence of straddle-converged path summations based on such parameterization curves with increasing length

$$L_0 = \mathcal{I}[\mathbf{u_0}, \mathbf{u_1}]$$

$$L_1 = \mathcal{I}[\mathbf{u_0}, \mathbf{u_1}, \mathbf{u_2}]$$

$$\vdots$$

$$L_n = \mathcal{I}[\mathbf{u_0}, \mathbf{u_1}, \mathbf{u_2}, \ldots, , \mathbf{u_i}, \ldots, , \mathbf{u_{2^n+1}}]$$

where $\mathbf{u_0} = \mathbf{u}_a$ and $\mathbf{u_{2^n+1}} = \mathbf{u}_b$ $\quad \forall n$.

In each step of this sequence, the parameterization curve is augmented with an additional point between each pair of points from the previous step. We call these points refinement points, and we call this sequence a refinement sequence.

Theorem 4.16. *A refinement sequence converges to a minimal of J.*

Proof. Given a chart G with metric M and with a bounded refinement sequence $L = L_0, L_1, \ldots, L_n$. Let $\mathbf{u}_{n,j}$ be a refinement point augmented in step $n+1$ of the sequence. From Definition 4.8 the piecewise path summation from step n is

$$\mathcal{L}[\mathbf{u}_{n,j-1}, \mathbf{u}_{n,j}, \mathbf{u}_{n,j+1}]$$

and the piecewise path summation from step $n+1$ is

$$\mathcal{L}[\mathbf{u}_{n+1,j-1}, \mathbf{u}_{n+1,j}, \mathbf{u}_{n+1,j+1}]$$

considering all such refinement points, then it follows from the triangle inequality that

$$L_n \leq L_{n+1} \quad \forall n$$

and so L is a monotonically increasing series and we can write

$$0 \leq L_0 \leq L_1 \leq \cdots \leq L_i \cdots \leq L_n < \infty$$

Let $\varepsilon > 0$, and let $J_L = \sup L$. Since $J_L - \varepsilon$ is not an upper bound for L, there exists N such that $L_N > J_L - \varepsilon$. Since L_n is increasing, we have $L_N \leq L_n$ for all $n \geq N$. Also $L_n \leq J_L$ for all n, so $n > N$ implies $J_L - \varepsilon < L_n \leq J_L$, and consequently

$$\lim_{n \to \infty} L_n = J_L$$

and finally, since each L_n is by Theorem 4.12 minimal, then J_L is minimal. $\quad\square$

4.3 The CVA algorithm Implementation

We have implemented the CVA algorithm as an importable Python library and have released this code as Free Open Source Software. The code is available through the Python Package Index (PyPI) as well as other downstream repositories. See Appendix A for installation details.

In Appendix B we include full length code listings for the most important files. This section explains the construction of the most important parts of the library.

4.3.1 Minpoint approximation

For a minpoint computation, we first need to assign a model G, with an attached metric M. Then we can call our minpoint function with a starting and an ending point.

In the first part of this code sequence we convert our points into an internal form and we find the dimensionality of our space.

```python
def minpoint(sa, sb):
    # we determine the dimension of our phase space by examining the sa
       point
    sa = np.asarray(sa)
    sb = np.asarray(sb)
    if np.allclose(sa,sb):
        return sa        # quick return if starting and ending are equal
    nparm = np.shape(sa)[0]
```

Now we construct a secant vector between the starting and ending points, and we take its center as our first approximation of a minpoint. We then want to find a trial

space, perpendicular to the secant vector, which includes this center point. The first step in this process is to find a non-orthogonal basis including the secant vector.

```python
# step 1: form a non-orthogonal basis space including our secant vector
basis = np.zeros((nparm,nparm))
cartesian_space = np.zeros((nparm,nparm))
for i in range(nparm):
    cartesian_space[i,i] = 1.0
# our non-orthogonal basis must include the primary (secant) unit
    vector
basis[0] = (sb - sa)/np.linalg.norm(sb - sa)
# we select a the set of cartesian unit vectors by eliminating the
    worst choice
worst_choice = np.argmax(np.abs(basis[0]))
# then we list the indices of the best cartesian choices
axis = [axis for axis in range(nparm) if axis != worst_choice]
for i in range(1,nparm):
    # and use them to form the rest of our basis
    basis[i] = cartesian_space[axis[i-1]]
    if np.allclose(basis[i],basis[0]):
        raise ValueError('colinear vectors found in basis')
```

Now that we have a non-orthogonal basis including the secant vector we can apply the Gramm-Schmidt process, or QR factorization, to convert to an orthonormal basis.

```python
# step 2: form an orthonormal basis including the secant vector
perp = np.empty((nparm,nparm))
perp[0] = basis[0]       # perp[0] is the secant unit vector
for i in range(1,nparm): # the Gramm-Schmidt summation
    perp[i] = basis[i]  # starting with the basis vector
    for j in range(i):  # then subtracting previous vector components
        perp[i] -= \
            perp[j]*np.inner(basis[i],perp[j])/np.inner(perp[j],perp[j])
    perp[i] = perp[i]/np.linalg.norm(perp[i]) # normalize
```

With a trial space in hand, and after setting an initial radius of interest, we are ready to use our metric and our model to find a metric-weighted path summation for

each trial point. The first part of this step is the setup of appropriate vectors to hold results.

```python
# step 3: starting with a midpoint and a radius, find the best path
sm = (sb-sa)/2.0 + sa
radius = np.linalg.norm(sb-sa)*_parms['tp_starting_trial_space_radius']
# our trial space has 5**(n-1) points,
# five points accross each axis centered on the current midpoint
    estimate
trial_shape = []
for i in range(nparm-1):
    trial_shape.append(5)
trial_array_shape = deepcopy(trial_shape)
trial_array_shape.append(nparm)
# trial_space = np.zeros(trial_array_shape) # the set of trial points
# trial_integral = np.zeros(trial_shape) # path summations for each
    trial point
maxtries = _parms['tp_max_trials']
```

Now that our setup is complete, we reach the inner loop of the minpoint approximation. Inside this loop we find path summations for each possible path from the starting point through a trial point and on to the ending point. We take that minimum path and use it as a new minpoint candidate. After reducing our radius of interest by 1/2 its previous value we continue with another repetition.

```python
while maxtries > 0 and radius > EPS*_parms['tp_eps_multiplier']:
    maxtries -= 1
    best_integral = np.infty
    # create a tuple of all possible trial index permutations
    trial_index_set = itertools.product(range(5),repeat=nparm-1)
    trial_space = np.zeros(trial_array_shape) # the set of trial points
    trial_integral = np.zeros(trial_shape) # path integrals for each
        trial point
    for it in trial_index_set: # loop over all trial permutations
        trial_space[it] = sm # our trial point includes the center
        for i in range(nparm-1): # and spans the basis coordinates
```

```
                trial_space[it] += radius*perp[i+1]*(it[i]-2)/2.0
                #print np.linalg.norm(radius*perp[i+1]*(it[i]-2)/2.0)
            trial_integral[it] = _parms['M'](sa,trial_space[it])+
                _parms['M'](trial_space[it],sb)
            if trial_integral[it] < best_integral:
                best_integral = trial_integral[it]
                bestit = it
        # prepare new trial (center and radius) based on our best path
        if best_integral < np.infty:
            sm = trial_space[bestit]
            radius = radius/2
        else:
            print "no solution found"
    return sm
```

After a suitable number of repetitions, we have found a best approximation for the minpoint, and there is nothing left to do but return it.

4.3.2 Straddling

The major challenge in implementing a straddling sequence is that we must pay close attention to indexing. An iteration consists of two parts. An odd straddle updates all odd numbered points with a new minpoint based on its two even numbered neighbors. Following that step, an even straddle updates all even numbered interior points with a new minpoint based on its two odd numbered neighbors. These two steps result in an improved path summation. By repeating this procedure, we arrive at a minimal polygonal path between the two endpoints.

```
def straddle(s, step):
    N = _parms['N']
    incr = int(N/(2**step))
    for iteration in range(_parms['tp_straddles']):
        for k in range(1, int(N/incr), 2):
            a = (k-1)*incr
```

```
        b = (k+1)*incr
        s[int(k*N/(2**step))] = minpoint(s[a], s[b])
    for k in range(2, int(N/incr), 2):
        a = (k-1)*incr
        b = (k+1)*incr
        s[int(k*N/(2**step))] = minpoint(s[a], s[b])
    return (s)
```

4.3.3 Refinement

The process of refinement involves augmenting the path with additional points, nearly doubling the sequence length with each step. Initially a solution vector is sized large enough to hold all steps ($2^{\text{steps}} + 1$). This very short code segment calls for a straddle convergence over successively longer subsequences.

```
def refine(s,steps):
    for step in range(1,steps+1):
        s = straddle(s,step)
    return s
```

4.4 Validation of results

In Section 4.2 we presented formal proofs validating the CVA algorithm. In this section we illustrate its validity through comparison of our solutions with some example cases for which we have known analytical solutions. We intend to show a high correlation between the analytical solutions and our numerical results.

The most basic example is finding the shortest path between two points on the surface of a unit plane, that being a line of length $\sqrt{2}$ [10].

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	1.414214	1.414214	0.0000
step 2	1.414214	1.414214	0.0000
step 3	1.414214	1.414214	0.0000
step 4	1.414214	1.414214	0.0000
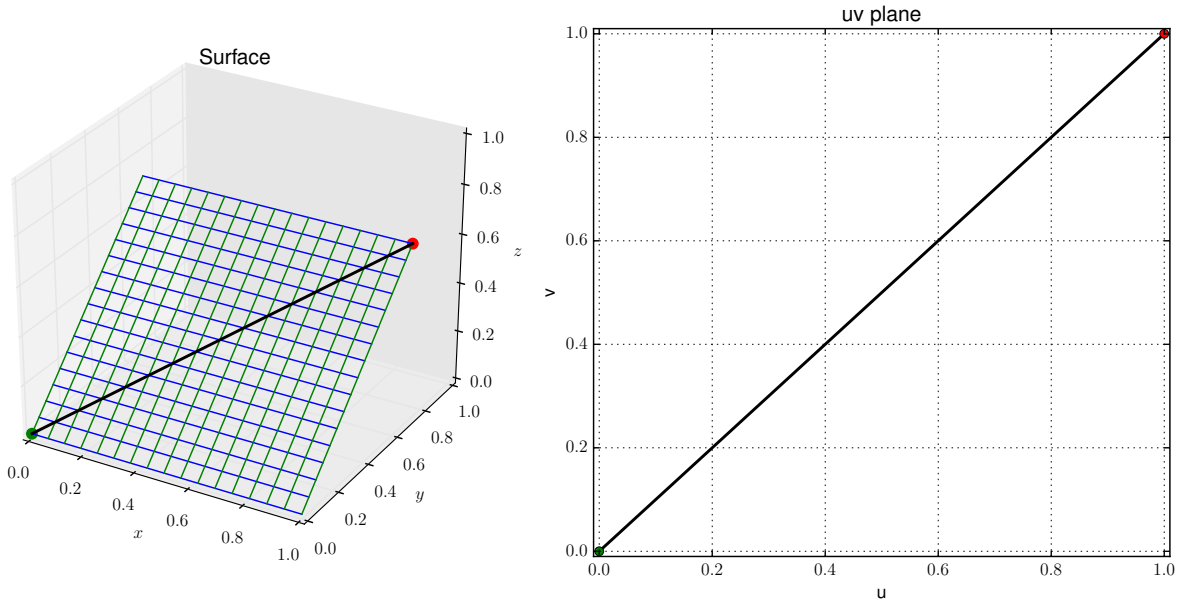
Figure 19: Tilted plane

Another longstanding known result is the minimum distance between two points on opposite sides of a unit cylinder. The cylinder is isometrically equivalent to a plane so Euclidean distance provides a result. In this example we have an analytical solution of $d = \sqrt{\pi^2 + 1}$. Figure 20 illustrates the results.

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	2.657971	3.296908	19.3799
step 2	3.216842	3.296908	2.4285
step 3	3.277726	3.296908	0.5818
step 4	3.292103	3.296908	0.1457
step 5	3.295707	3.296908	0.0364
step 6	3.296608	3.296908	0.0091
step 7	3.296834	3.296908	0.0023
step 8	3.296890	3.296908	0.0006
step 9	3.296904	3.296908	0.0001
step 10	3.296908	3.296908	0.0000
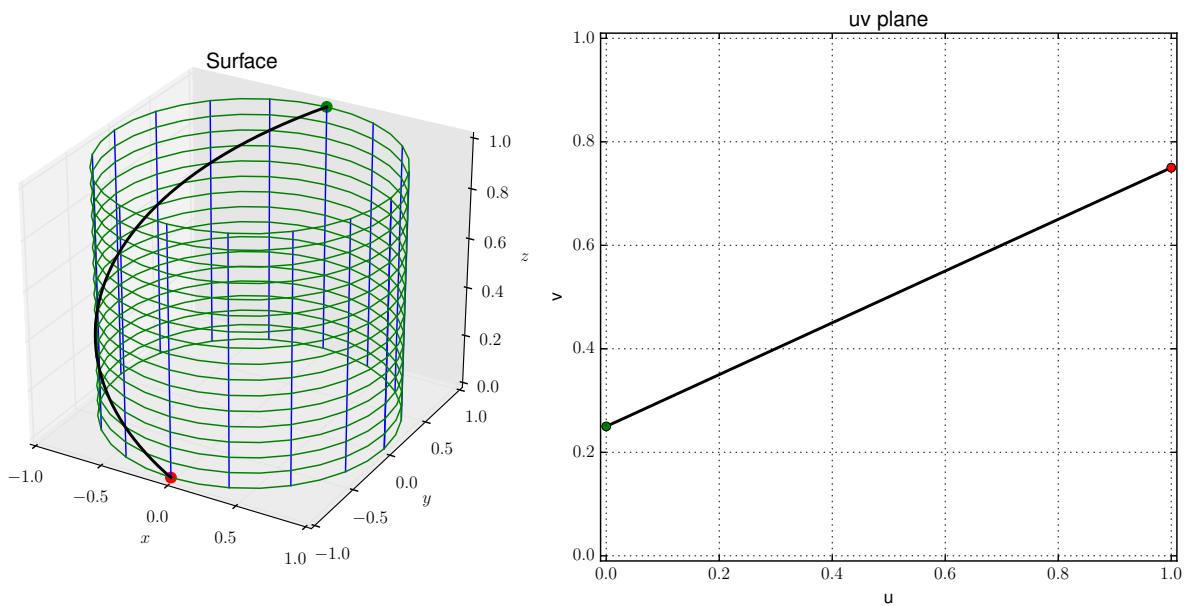
Figure 20: Cylinder

The hyperboloid has, interestingly enough, straight lines as its geodesics (for suitably chosen paths). The analytical solution therefore involves finding the length of a line between two points in 3-dimensional space for such a suitably chosen path.

For the points $\mathbf{u}_a = \langle u_a, v_a \rangle = (1.0, 0.0)$ and $\mathbf{u}_b = \langle u_b, v_b \rangle = (0.0, , 0.36024032)$, we define φ and θ to depend on u and v as

$$\varphi_a = 3(0.5 - u_a)$$

$$\varphi_b = 3(0.5 - u_b)$$

$$\theta_a = 2\pi(v_a - 0.5)$$

$$\theta_b = 2\pi(v_b - 0.5)$$

so the endpoints are thus given by

$$\mathbf{x}_a = (\cosh(\varphi_a)\cos(\theta_a), \cosh(\varphi_a)\cos(\theta_a), \sinh(\varphi_a))$$

$$\mathbf{x}_b = (\cosh(\varphi_b)\cos(\theta_b), \cosh(\varphi_b)\cos(\theta_b), \sinh(\varphi_b))$$

and the distance between them by

$$d = \operatorname{norm}(\mathbf{x}_b - \mathbf{x}_a)$$

which yields an analytical solution of 6.022512. Figure 21 illustrates the results.

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	6.022512	6.022512	0.0000
step 2	6.022512	6.022512	0.0000
step 3	6.022512	6.022512	0.0000
step 4	6.022512	6.022512	0.0000
step 5	6.022512	6.022512	0.0000
step 6	6.022512	6.022512	0.0000
step 7	6.022512	6.022512	0.0000
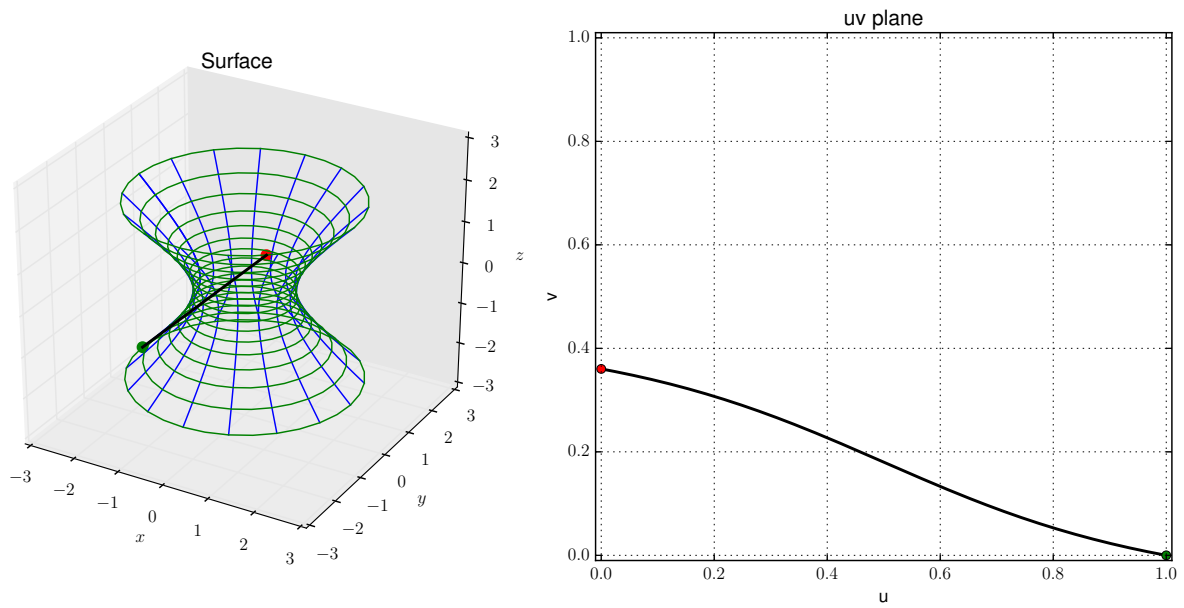
Figure 21: Hyperboloid

The cosh-shaped surface is isometrically equivalent to a plane so its geodesic is also a Euclidean path. For the points

$$\mathbf{u}_a = (0.1, \sinh^{-1}(0.5 \cdot 0.1))$$

$$\mathbf{u}_b = (1.0, \sinh^{-1}(0.5 \cdot 1.0))$$

we have an analytical solution of $d = 0.9\sqrt{1 + 0.5^2}$. Figure 22 illustrates the results.

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	1.005871	1.006231	0.0358
step 2	1.006141	1.006231	0.0089
step 3	1.006208	1.006231	0.0022
step 4	1.006225	1.006231	0.0006
step 5	1.006229	1.006231	0.0001
step 6	1.006230	1.006231	0.0000
step 7	1.006231	1.006231	0.0000
step 8	1.006231	1.006231	0.0000
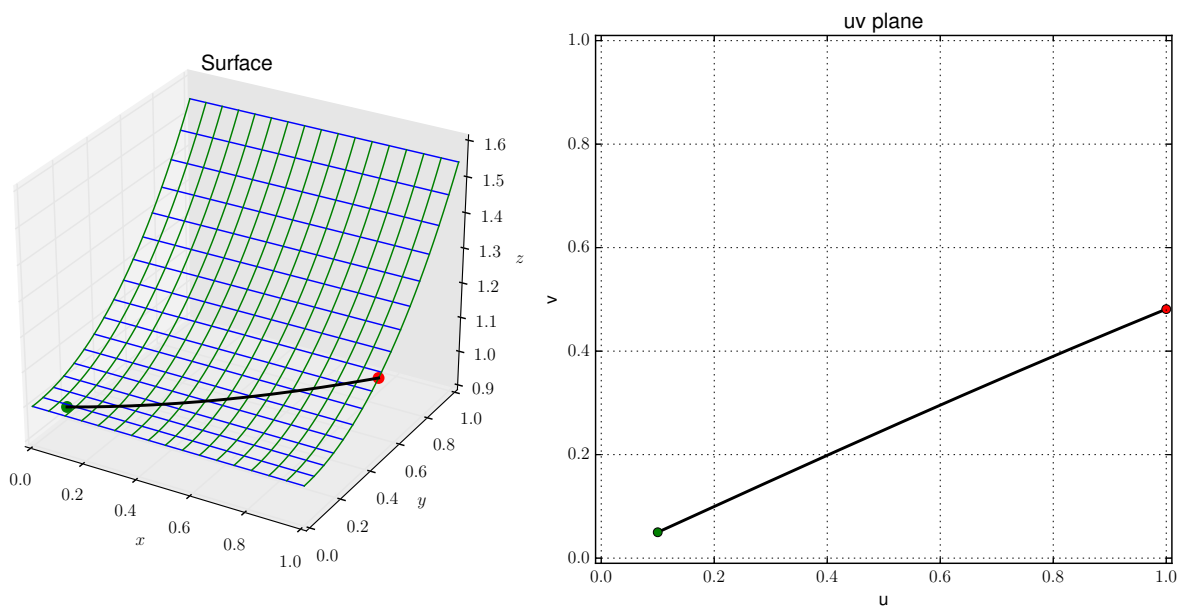
Figure 22: Cosh-shaped surface

Geodesics on the sphere have been of interest to navigation for hundreds of years. We first consider a path on the unit sphere from a starting point to an ending point directly opposite. We would expect this geodesic to have a length of exactly π.

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	2.815178	3.141593	10.3901
step 2	3.061408	3.141593	2.5524
step 3	3.121238	3.141593	0.6479
step 4	3.136483	3.141593	0.1627
step 5	3.140314	3.141593	0.0407
step 6	3.141273	3.141593	0.0102
step 7	3.141513	3.141593	0.0025
step 8	3.141573	3.141593	0.0006
step 9	3.141588	3.141593	0.0002
step 10	3.141591	3.141593	0.0000
step 11	3.141592	3.141593	0.0000
step 12	3.141593	3.141593	0.0000
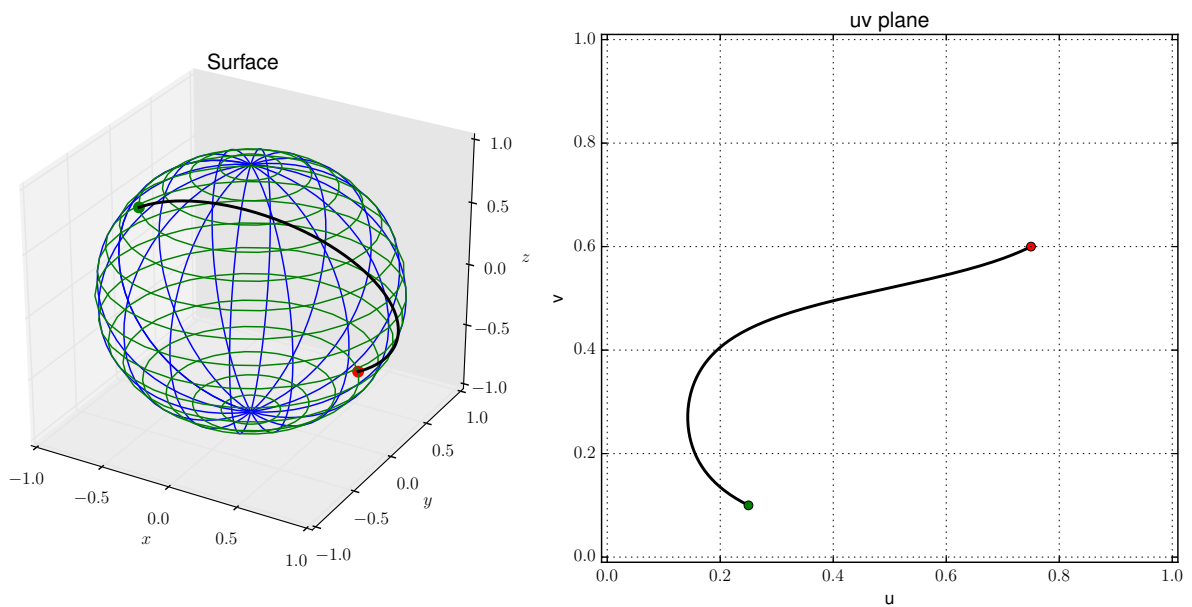
Figure 23: Unit Sphere

As a final example we again take the unit sphere and find a geodesic between two closer points, this time expressing these points in latitude/longitude format. The analytical solution follows directly from the law of cosines in spherical trigonometry.

```
sa = cva.model.latlon(30, -90); sb = cva.model.latlon(45, -75)
theta = 15.0*pi/180; phi_a = pi/3; phi_b = pi/4
d = acos(cos(phi_b)*cos(phi_a) + sin(phi_b)*sin(phi_a)*cos(theta))
```

Refinement	CVA approximation	analytical solution	difference (percent)
step 1	0.332634	0.333019	0.1155
step 2	0.332923	0.333019	0.0289
step 3	0.332995	0.333019	0.0072
step 4	0.333013	0.333019	0.0018
step 5	0.333018	0.333019	0.0005
step 6	0.333019	0.333019	0.0001
step 7	0.333019	0.333019	0.0000
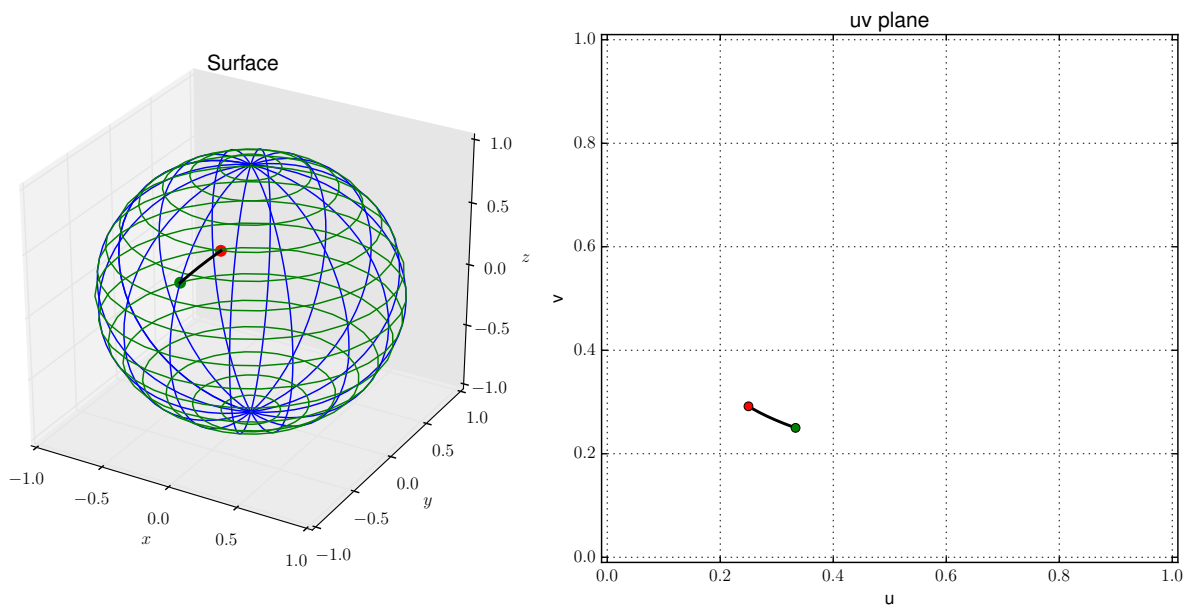
Figure 24: Unit Sphere

4.5 Additional examples

At this point we have validated the CVA algorithm both with mathematical proofs and by comparison with known solutions. We can now move on to explore minimal paths on surfaces and with metrics many of which have no known analytical solution.

4.5.1 Example: Torus
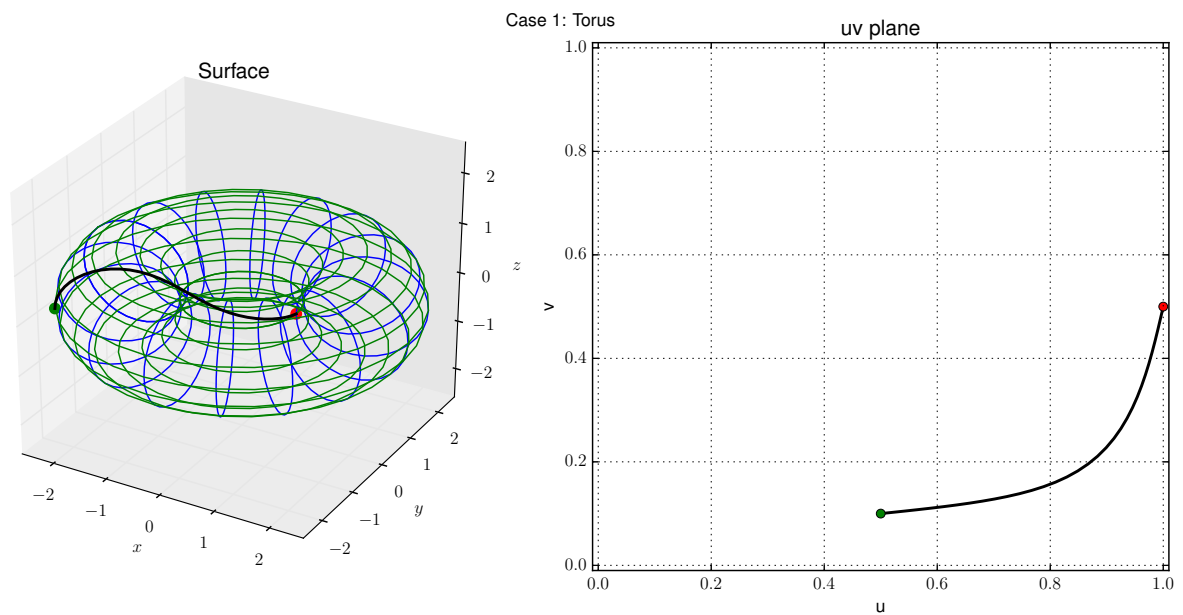
Figure 25: Torus

4.5.2 Example: Earth WGS84 Reference Model
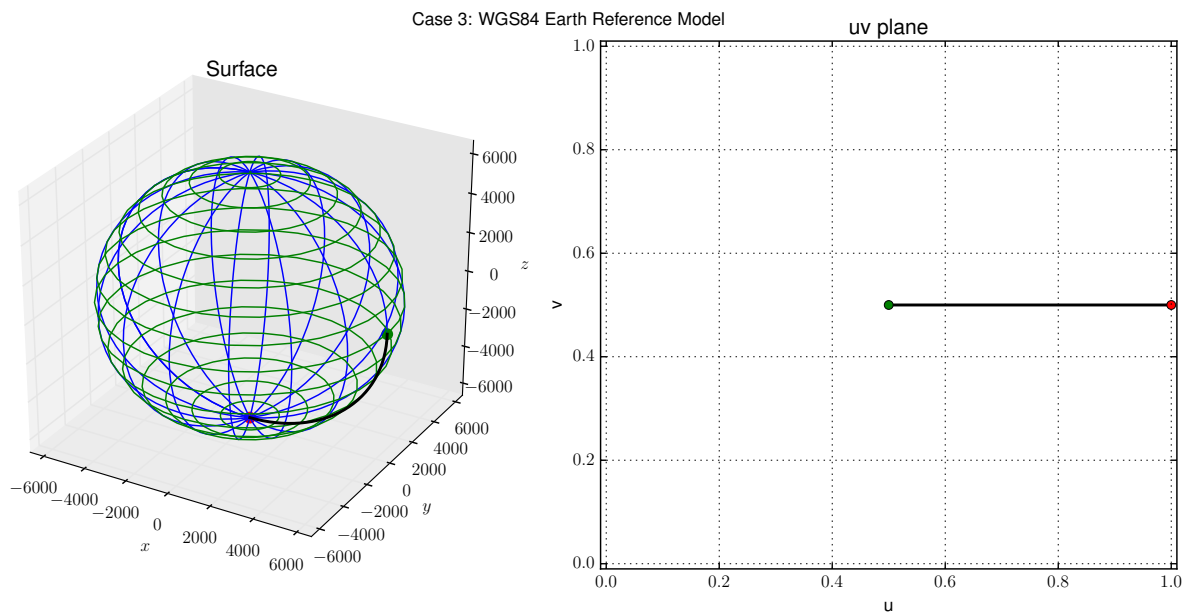
Figure 26: Earth WGS84 Ellipsoidal Model

4.5.3 Example: Sphylinder

We consider a surface which morphs between the shape of a sphere at one limit and a cylinder at the other limit. We call this surface a sphylinder in recognition of its two limiting shapes. We define the mapping for the sphylinder as follows. Let our u parameter represent a "latitude" ranging from $u = 0$ (north pole) to $u = 1$ (south pole), and let v represent a "longitude" ranging from $v = 0$ to $v = 1$ (2π). Next, convert u, v into φ, θ as

$$\varphi = \pi u$$

$$\theta = 2\pi v$$

and let p be a "morphing parameter" where $p = 1$ models a sphere and $p \to 0$ approaches the model of a cylinder. For this example we set

$$p = 0.5$$

and model the surface as

$$\mathbf{x} = (x, y, z)$$

$$x = (\sin \varphi)^p \cos \theta$$

$$y = (\sin \varphi)^p \sin \theta$$

$$z = |\cos \varphi|^p \text{ if } \varphi < \pi/2$$

$$z = -|\cos \varphi|^p \text{ if } \varphi \geq \pi/2.$$

Figure 27 shows a geodesic on the sphylinder.
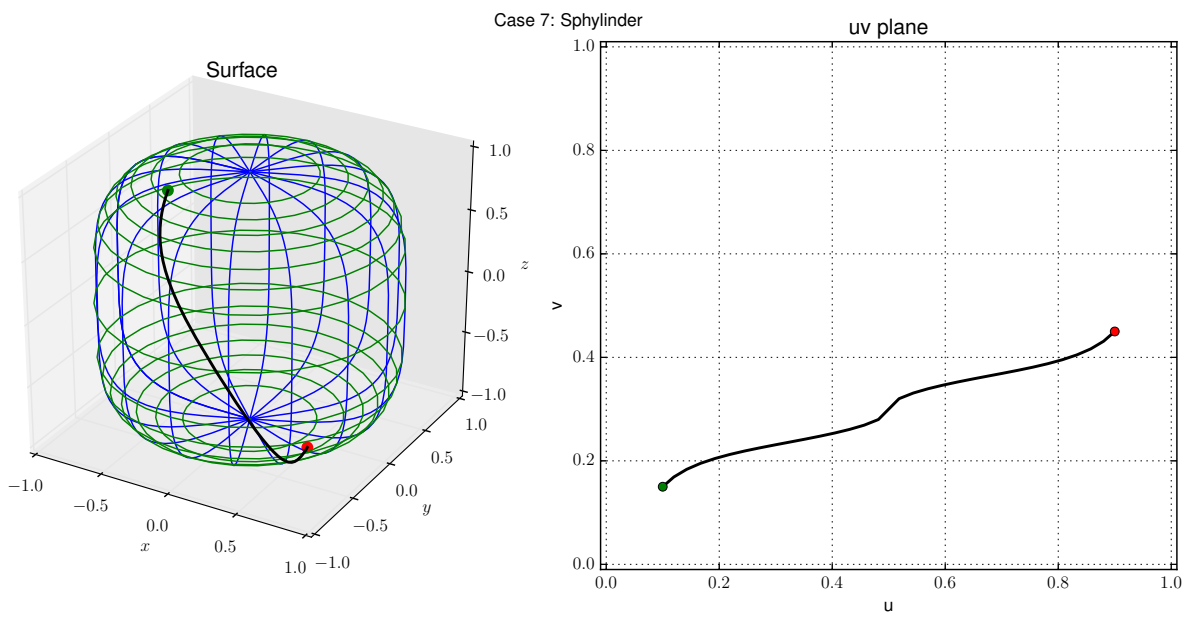
Figure 27: Sphylinder

4.5.4 Example: Capped Cylinder
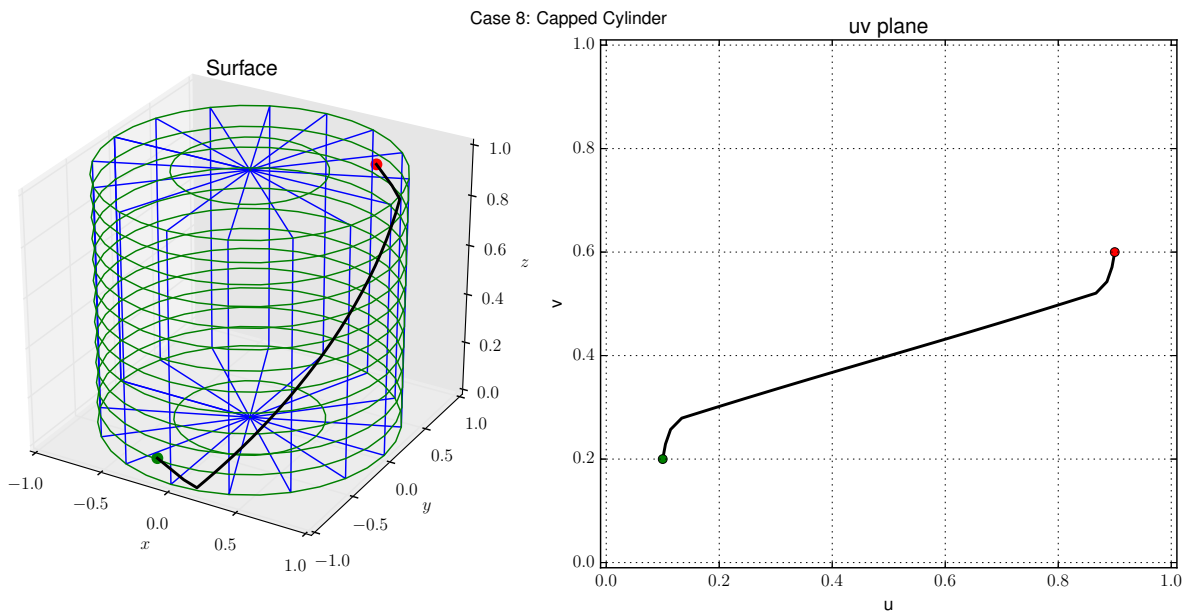
Figure 28: Capped Cylinder

69

4.5.5 Example: Moebius Strip
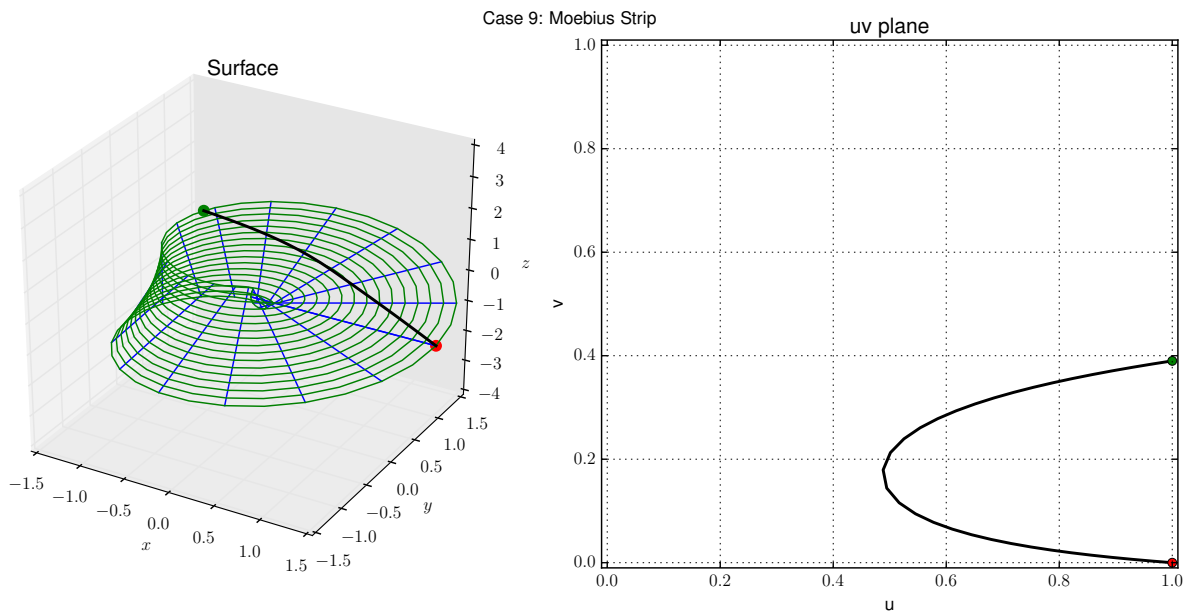
Figure 29: Moebius Strip

4.5.6 Example: Brachistochrone in Earth Gravity

Recall the brachistochrone from Section 3.1.2.3. Here we apply the CVA algorithm with the model of the plane and a brachistochrone metric to obtain an approximation to the cycloid solution.
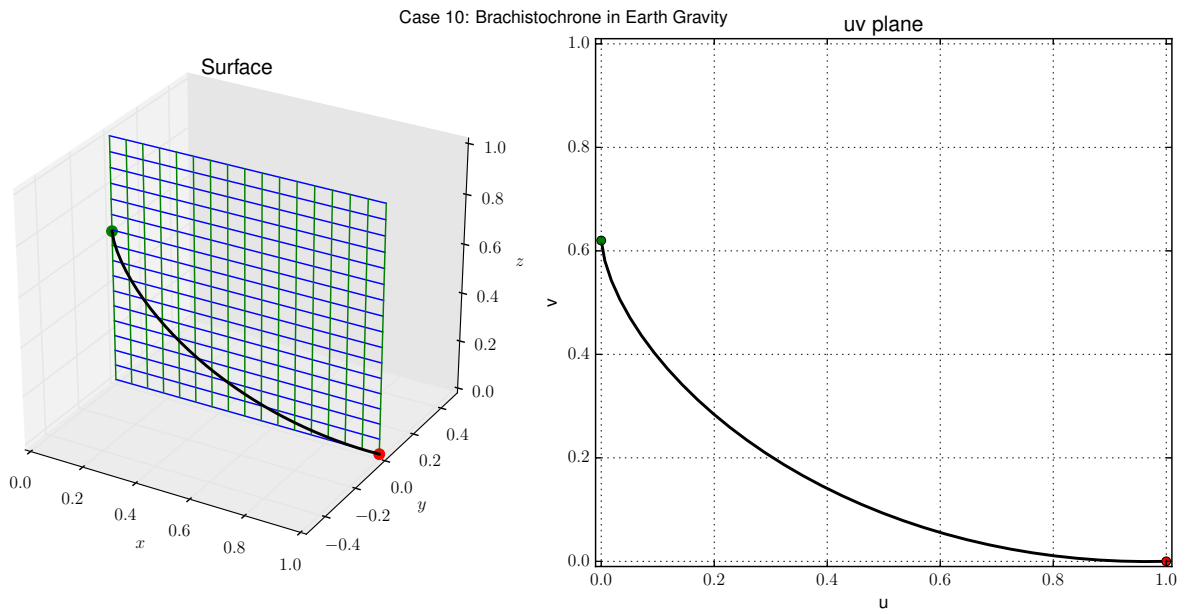
Figure 30: Brachistochrone in Earth Gravity

4.5.7 Example: Brachistochrone on a Tilted Plane

We verify that tilting the plane does not alter the brachistochrone solution.
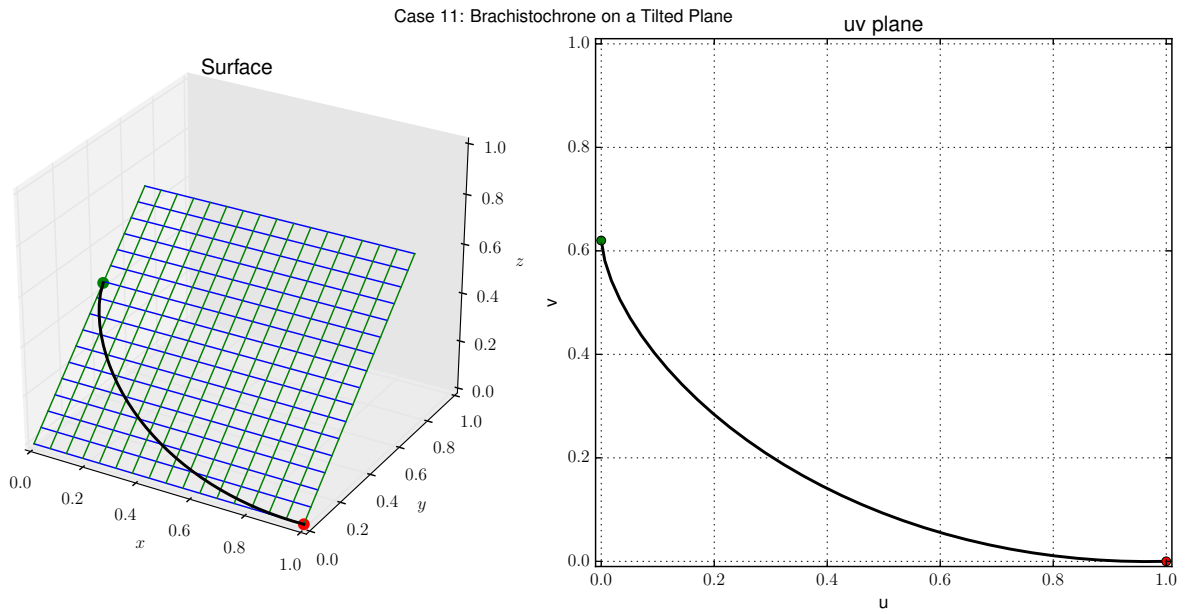
Figure 31: Brachistochrone on a Tilted Plane

4.5.8   Example: Brachistochrone in Moon Gravity

The time taken to traverse the optimal curve in Moon gravity is greater than that obtained in Earth gravity, but here we show that the shape of the curve remains the cycloid as we would expect.
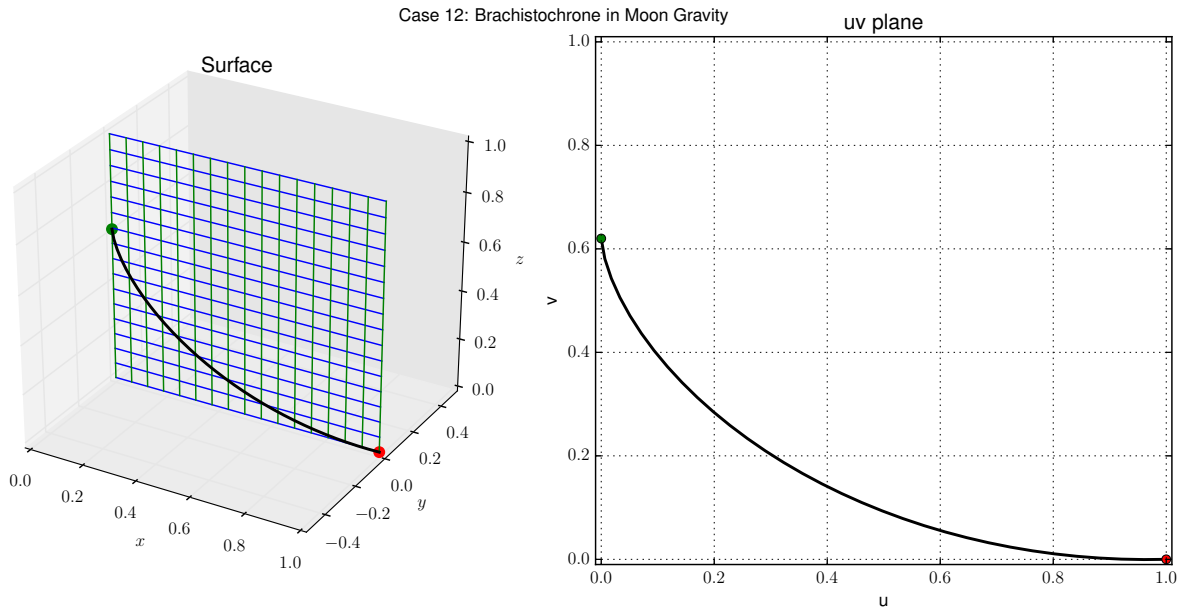
Figure 32: Brachistochrone in Moon Gravity

4.5.9 Example: Brachistochrone on a Unit Sphere

The CVA algorithm can be used to answer new questions that have up to this point evaded analytical solution. Here we ask for the path constrained to the surface of a unit sphere that would minimize the time for a particle to descend from a starting point (at rest) to an ending point under the influence of (negative z oriented) Earth gravity. Figure 33 illustrates the result of a CVA solution of the brachistochrone metric on a unit sphere surface model.
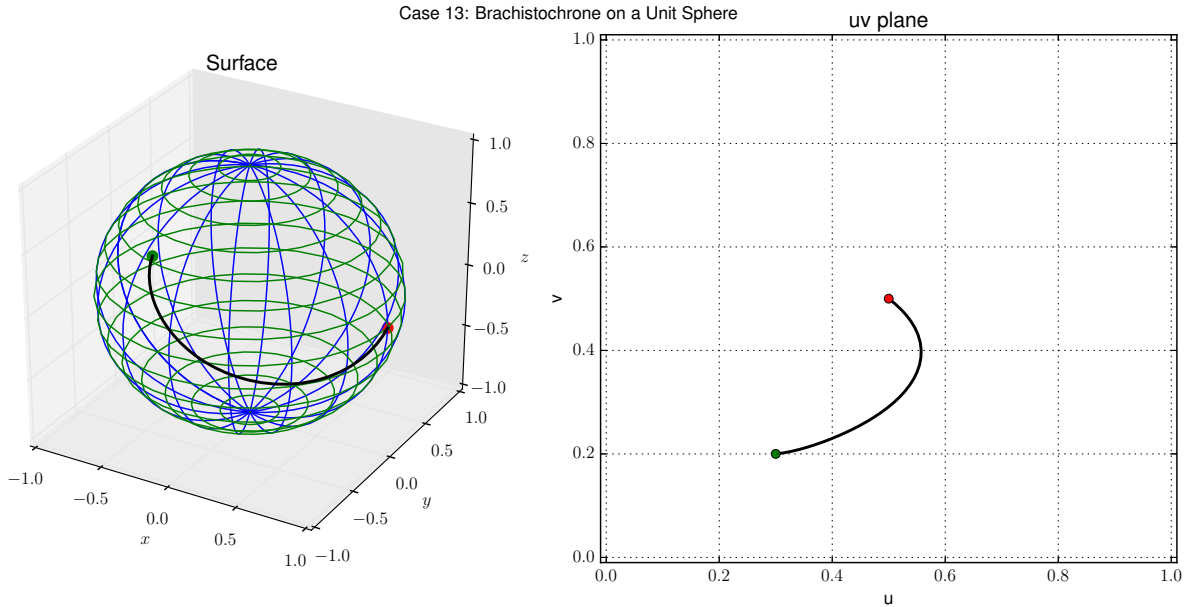
Figure 33: Brachistochrone on a Unit Sphere

4.5.10 Example: Brachistochrone on a Hyperboloid

Similarly we can ask for the path constrained to the surface of a hyperboloid that would minimize the time for a particle to descend from a starting point (at rest) to an ending point under the influence of (negative z oriented) Earth gravity. It is interesting to observe that the best "strategy" for the particle is to descend near the vertical as it begins its acceleration, then increasing its angle of attack in the later phase of its journey. Figure 34 illustrates the result of a CVA solution of the brachistochrone metric on a hyperboloid surface model.
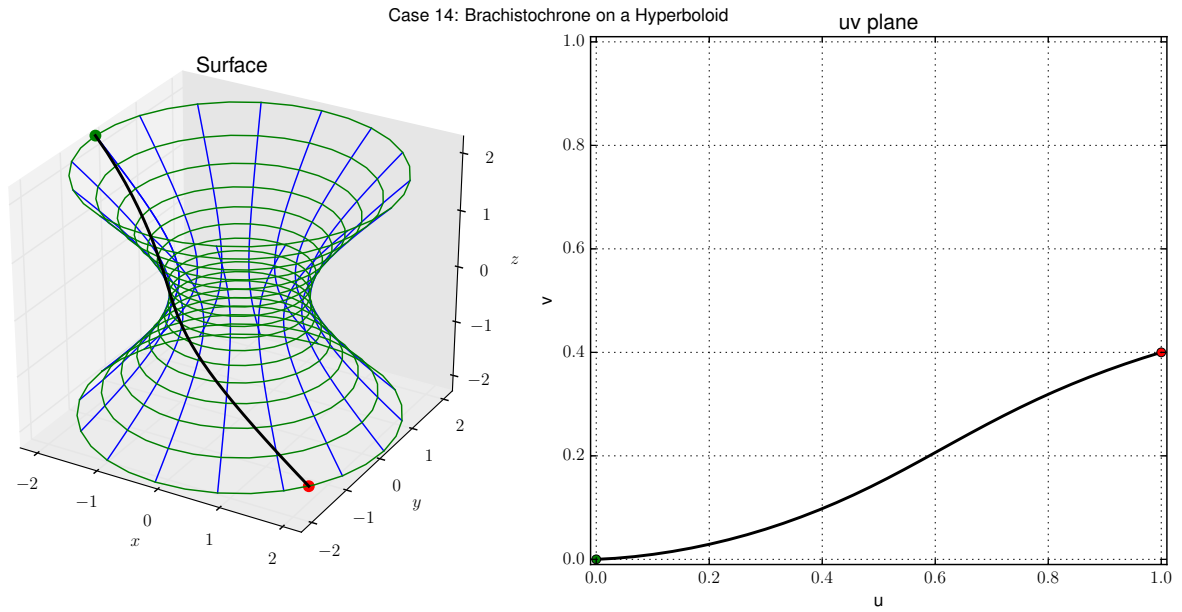
Figure 34: Brachistochrone on a Hyperboloid

4.6 Extension to higher order spaces

When moving from $\mathbb{R}^3$ to higher dimensions, it becomes convenient to first adopt a suitable notation. Let $\mathbf{u} = (u_0, u_1, \ldots, u_{m-1})$ represent a point in a parameter space in $\mathbb{R}^m$, and let $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})$ represent a point in $\mathbb{R}^n$, $\quad m < n$.

Up to this point our primary metric has been the Euclidean distance metric in $\mathbb{R}^3$ which we expressed, in the notation of differential forms, as

$$ds^2 = dx^2 + dy^2 + dz^2.$$

As we now move into $\mathbb{R}^n$ we extend the distance metric to its more general form

$$ds^2 = \sum_{i=0}^{n-1} dx_i^2$$

and specifically in $\mathbb{R}^4$ two additional metrics take on special importance.

4.6.1 Minkowski metric

The Minkowski metric applies to flat spacetime, in that it provides a measure between events occurring in a region of spacetime void of mass. The Minkowski metric describes the essence of special relativity.

$$ds^2 = d(ct)^2 - dx^2 - dy^2 - dz^2$$

We give this metric here in rectangular coordinates with time expressed as ct with units of light seconds.

4.6.2 Schwarzschild metric

The Schwarzschild metric applies to curved spacetime in that it provides a measure between events occurring in a region of spacetime containing a central mass. The Schwarzschild metric describes the essence of general relativity. We give this metric here in spherical coordinates with time expressed as ct with units of light seconds.

$$c^2 ds^2 = \left(1 - \frac{2GM}{rc^2}\right) d(ct)^2 - \frac{1}{1 - \frac{2GM}{rc^2}} dr^2 - r^2 d\theta^2 - r^2 \sin^2\theta d\varphi^2$$

If we wish to express this metric in Cartesian coordinates, we must add the relationships

$$x = r \sin \theta \cos \varphi$$

$$y = r \sin \theta \sin \varphi$$

$$z = r \cos \theta$$

$$ct = ct$$

4.6.3 Viewing higher dimensions

We exist in a 3-dimensional world and therefore our intuition into higher dimensions is limited. In order to view our results in a meaningful way we borrow a convention from the architects and engineers. They typically represent 3-dimensional objects in the form of orthographic projections. Specifically, three 2-dimensional views (top view, front view, side view) represent the 3-dimensional object. We can extend this idea to view 4-dimensional objects with four 3-dimensional views, taking each view as a slice of our 4-dimensional object with one axis set to zero. A 3-dimensional parameter space is easily represented in a single 3-dimensional view.

In dimensions higher than four, we can continue to extend this view principle by recognizing that $\binom{n}{n-3} = \frac{n!}{3!(n-3)!}$ combinations will suffice to view a space of dimension n.

dimensionality of our space	number of 3-d views required
3	1
4	4
5	10
6	20
7	35
8	56
9	84
10	120

The CVA library handles the general case in that it can generate views of arbitrary dimension, and we have tested through 7-dimensional spacial views.

4.6.4 Spherical model in $\mathbb{R}^n$

As a test of the CVA algorithm in higher dimensional spaces, we have chosen the hypersphere as our model. We extend an $\mathbb{R}^3$ sphere to a general $\mathbb{R}^n$ hypersphere by adding additional latitude parameters as follows.

```python
def sphere(s):
    s,x = startmodel(s)

    # start of model mapping
    n = np.shape(s)[1]
    length = np.shape(s)[0]
    # Convert to rectangular coordinate system
    prod = 1.0
    for i in range(n-1):
        x[:,n-i] = prod * np.cos(np.pi * s[:,i])
        prod = prod * np.sin(np.pi * s[:,i])
    x[:,0] = prod * np.cos(2*np.pi * (s[:,n-1]-0.5))
    x[:,1]  = prod * np.sin(2*np.pi * (s[:,n-1]-0.5))
    # end of model mapping

    x = finishmodel(x,s)
    return x
```

As our first example we find a geodesic on a hypersphere in $\mathbb{R}^4$. Figure 35 illustrates the result of a CVA solution of the generalized Euclidean distance metric on a hypersphere surface model. The surface is depicted as a set of four 3-dimensional slices each obtained by setting one of the four dimensions to zero. The geodesic is depicted in the *uvw* parameterization space and also in each hypersurface slice. On viewing the result, it is interesting to conjecture that a geodesic on a hypersphere lies on a plane intersecting the origin.
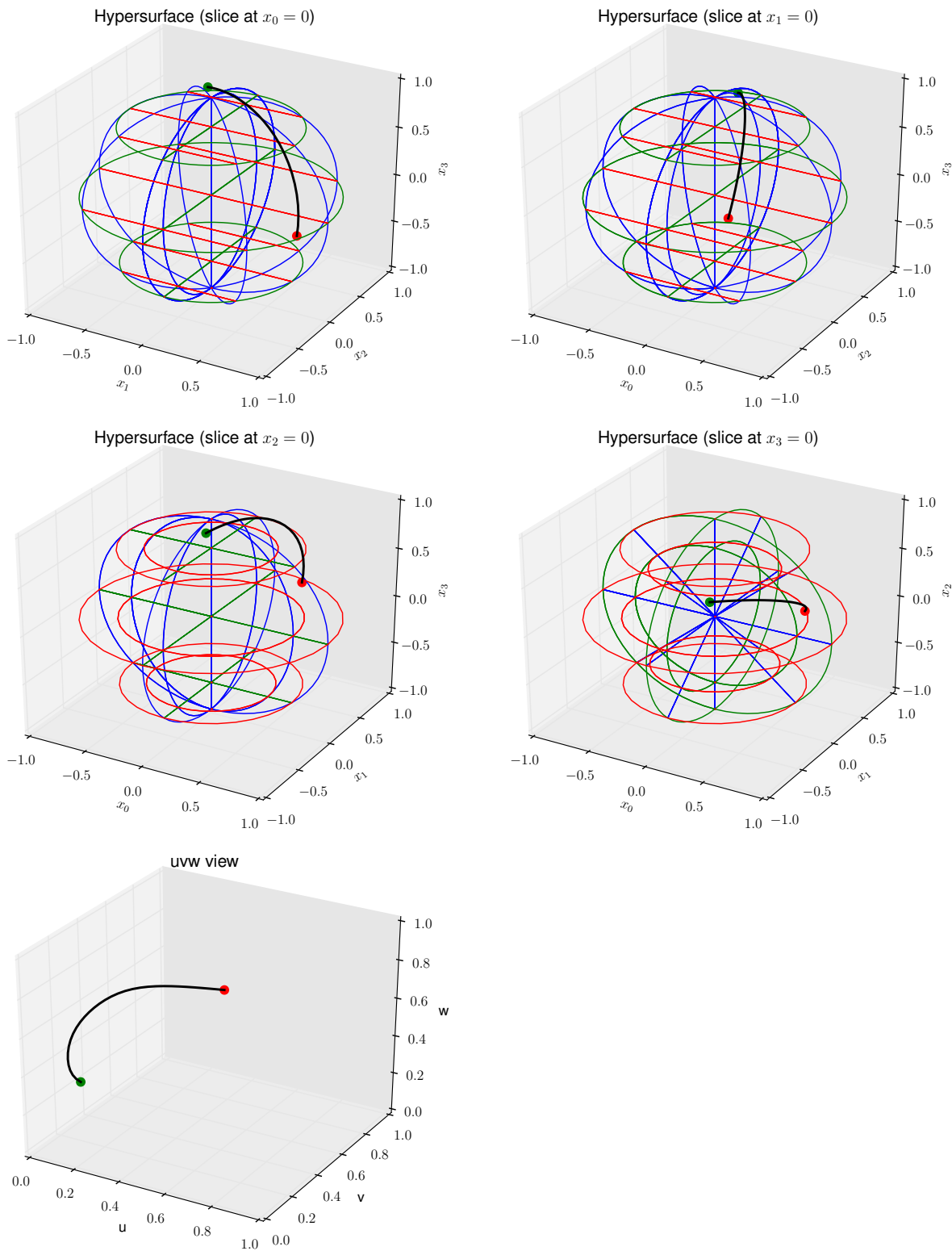
Figure 35: Hypersphere in $\mathbb{R}^4$

4.6.6 Example: Inflating Sphere in Spacetime

To further explore the application of the CVA algorithm to higher dimensional spaces, we model an inflating sphere in 4-dimensional spacetime. The sphere is inflating at a rate of 50% of light speed. To this model we attach the Minkowski metric. The CVA solution on this model and with this metric depicts the shortest path of a particle possessed with unconstrained speed as it moves between a starting event and an ending event in flat spacetime. Figure 36 illustrates the result of a CVA solution of the Minkowski metric on an inflating sphere surface model. When interpreting the result it is interesting to notice that the test particle's best strategy is to favor spacial movements (at near lightspeed) while the sphere is yet small then favor time movements (waiting with minimal spacial movement) as the inflation nears its goal.
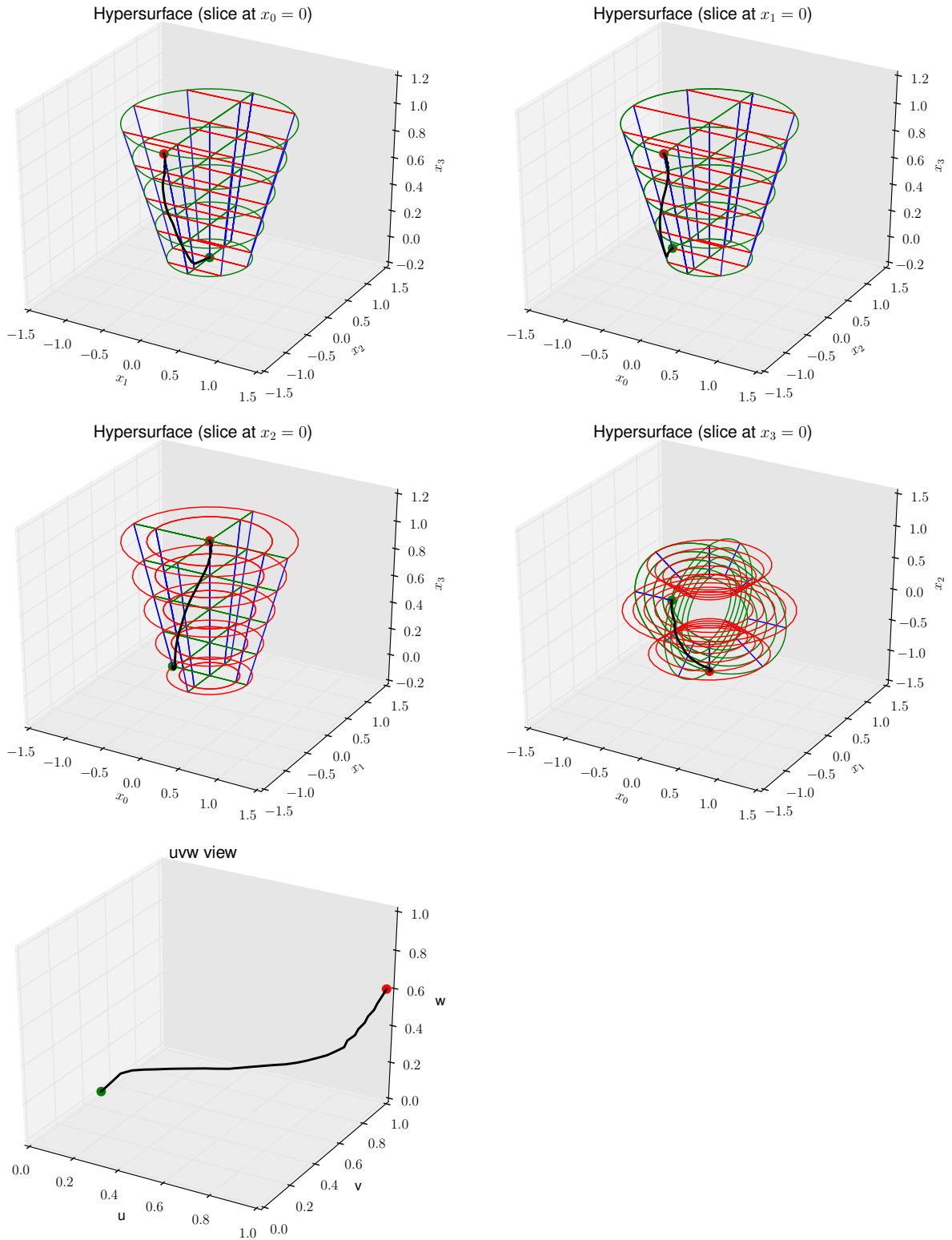
Figure 36: Inflating Sphere in Spacetime

4.6.7   Example: Collapsing Sphere in Spacetime

Similar to the previous example we model a collapsing sphere in 4-dimensional spacetime. The sphere is collapsing at a rate of 50% of light speed. To this model we attach the Minkowski metric. The CVA solution on this model and with this metric depicts the shortest path of a particle possessed with unconstrained speed as it moves between a starting event and an ending event in flat spacetime. Figure 37 illustrates the result of a CVA solution of the Minkowski metric on a collapsing sphere surface model. When interpreting the result and compare it to the previous result, we notice that the test particle's best strategy in this case is to favor time movements while the sphere is large and to delay favoring spacial movements (at near lightspeed) until the sphere becomes small.
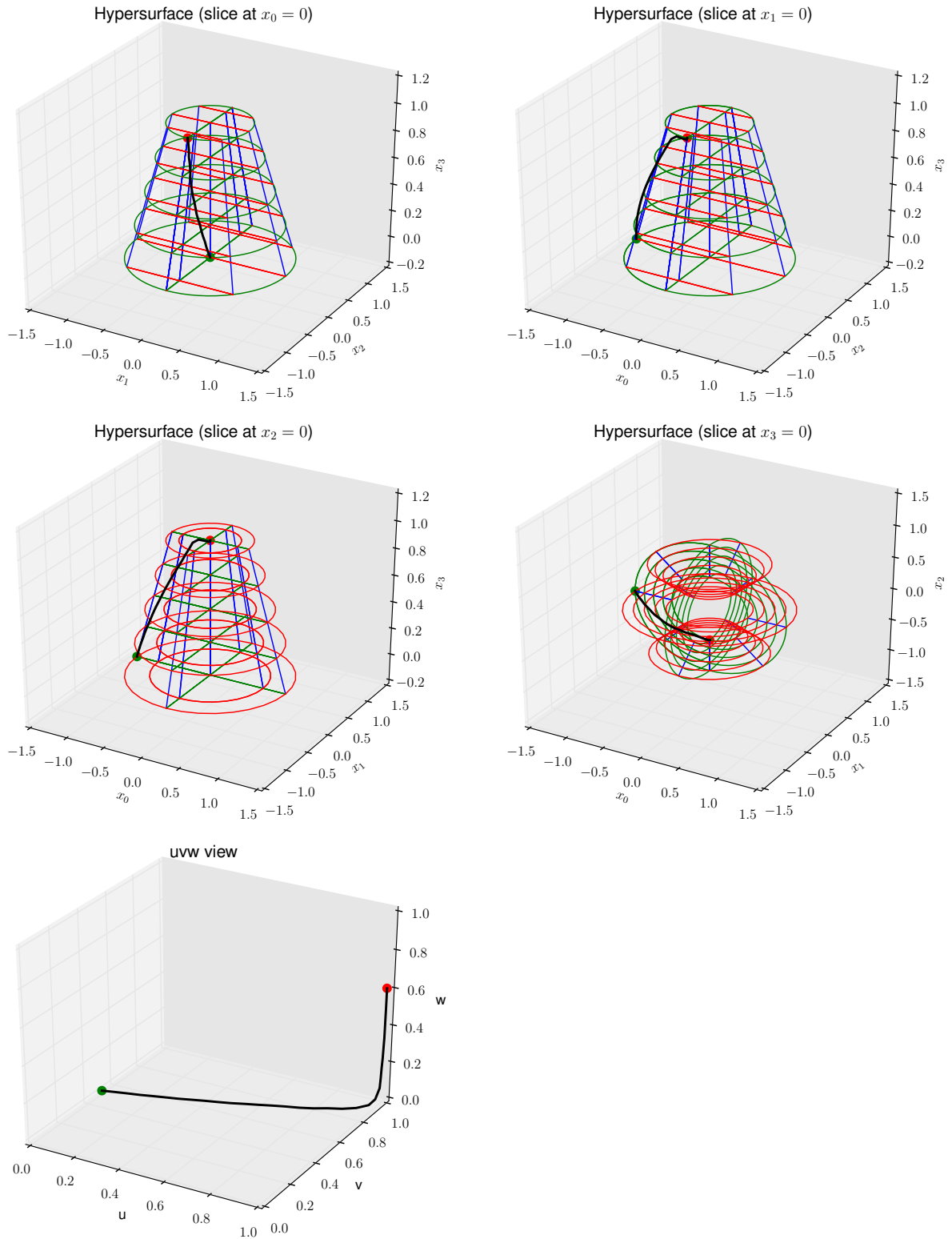
Figure 37: Collapsing Sphere in Spacetime

4.6.8   Example: Geodesic Family near a Black Hole

We explore the Schwarzschild metric (curved spacetime) by attaching it to a plane with a massive object at its center. In this model we assume the mass of Sgr A*, the Milky Way's supermassive black hole, and model a region of space measuring 10 event horizons in each direction. Minimal paths in this model and with this metric correspond to the paths a photon could take as it passes through this region of space. Figure 38 illustrates the result of a CVA solution of the Schwarzschild metric on a planar surface model.
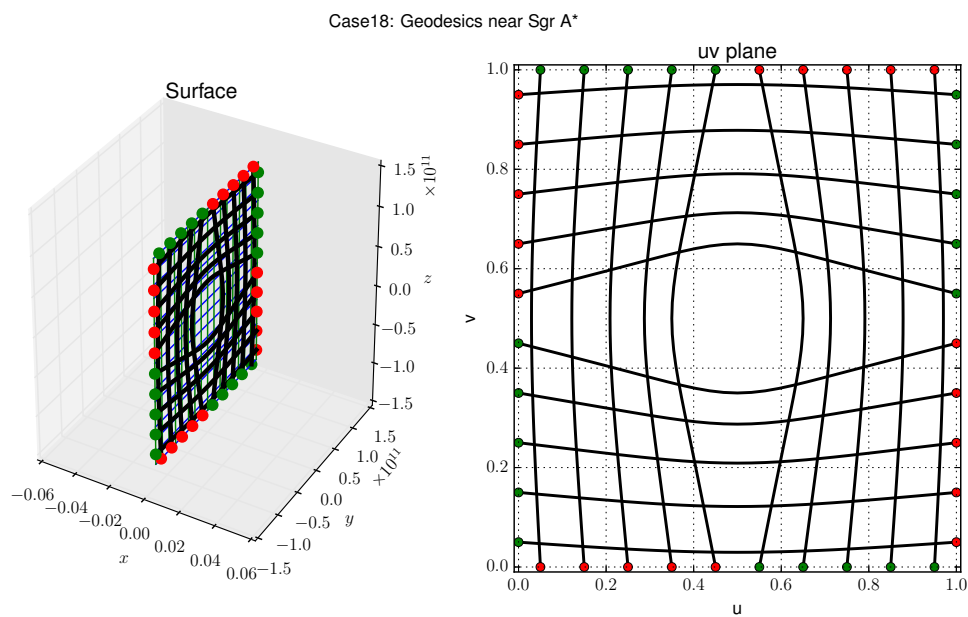
Figure 38: Geodesics in Spacetime with the Schwarzschild Metric

4.6.9 Example: Hypersphere in $\mathbb{R}^5$

As our final example in Figures 39 and 40 we show a geodesic on a hypersphere in $\mathbb{R}^5$. Notice that there are a total of 10 combinations of 3-dimensional slices representing a 5-dimensional object. Also, we are working with a 4-dimensional parameter space which requires four 3-dimensional views.
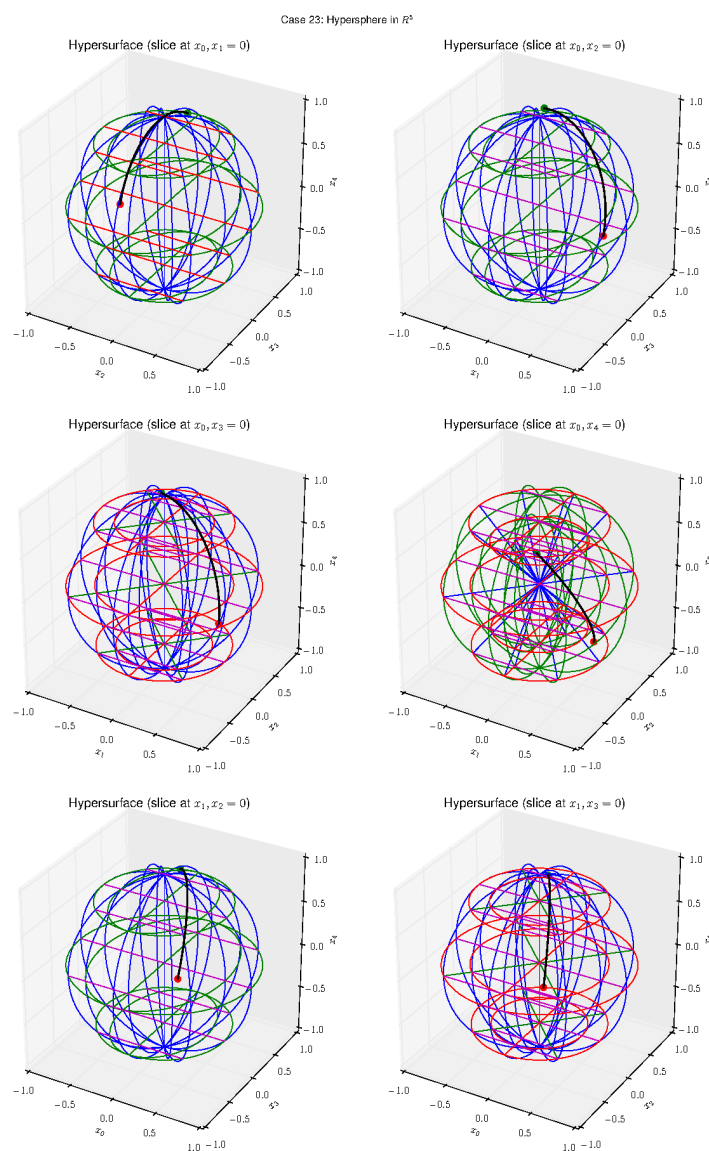
Figure 39: Hypersphere in $\mathbb{R}^5$
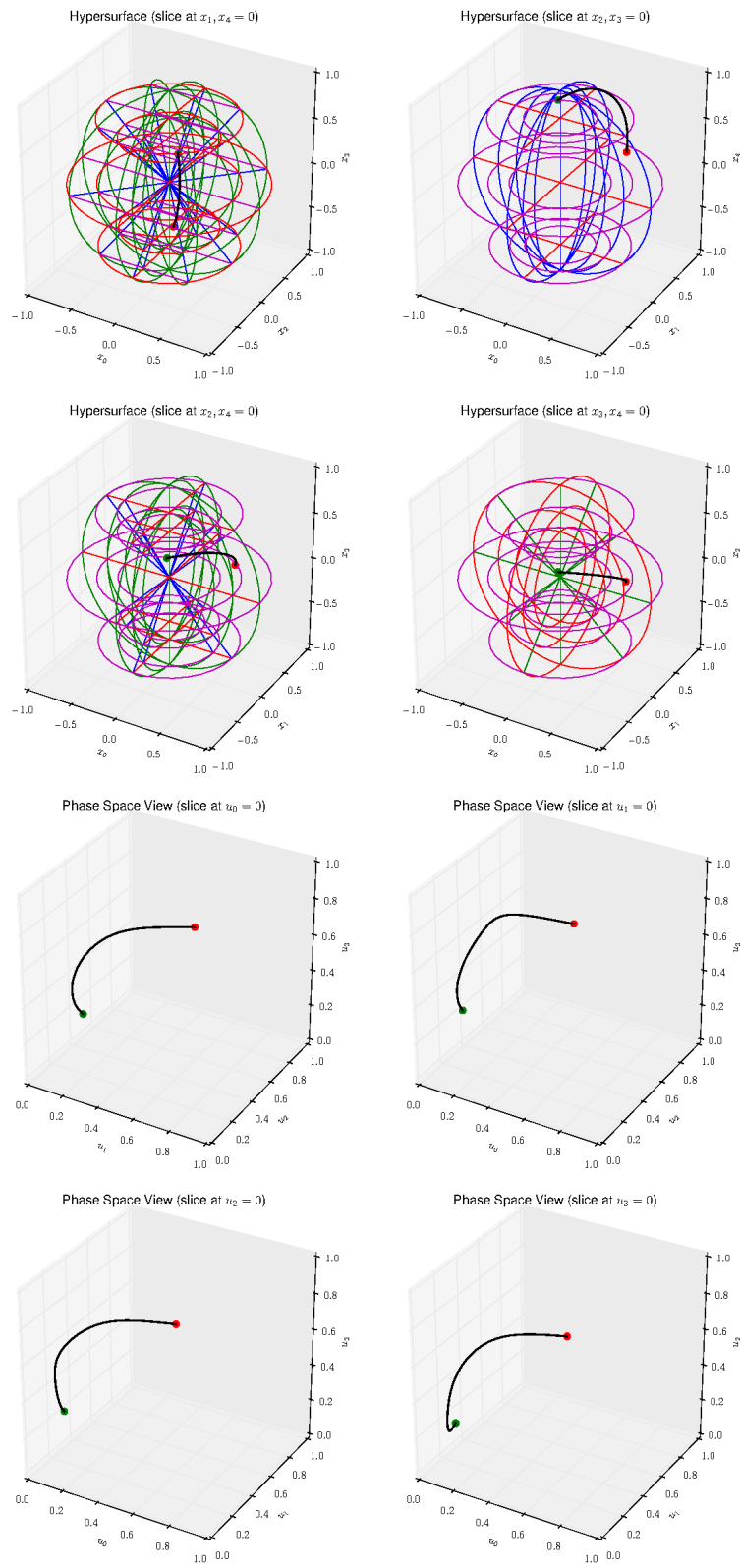
Figure 40: Hypersphere in $\mathbb{R}^5$

5 FUTURE DIRECTIONS

5.1 Observation 1

Considering further the relationship between the CVA algorithm and classical calculus of variations, we note that what we have called in this paper a *metric* is identified with the term *objective functional*, and what we have called a *surface* with starting and ending points corresponds to a set of boundary conditions. We proved in Theorem 4.16 that the CVA solution converges to the minimal path integral, and thus it follows that it also approximates a solution of the set of differential equations of the kind known in classical calculus of variations as Euler's equation [6]. We state here as a conjecture, that the CVA algorithm can be applied to the numerical approximation of a class of partial differential equations with prescribed boundary conditions. We propose an investigation in this direction to expand the generality of this observation.

5.2 Observation 2

In this thesis the CVA algorithm is described in the context of an m-manifold from which we take a single chart. The generalization of the CVA algorithm to comprehend all charts in the atlas should follow directly from the observation that charts are required to share points with their neighbors and that a metric on an m-manifold is required to be independent of the selected chart. It is therefore possible to introspect the geometry of an m-manifold and thereby to implement the selection of an appropriate chart on which the minpoint approximation will be valid or even optimal.

BIBLIOGRAPHY

[1] Ralph Abraham, Jerrold E Marsden, and Jerrold E Marsden. *Foundations of mechanics.* Benjamin/Cummings Publishing Company Reading, Massachusetts, 1978.

[2] Vladimir Igorevich Arnold. *Mathematical methods of classical mechanics*, volume 60. Springer Science & Business Media, 1989.

[3] CT Chen, EM Benglas, SK Singh, DC Bullock, and R Whitinger. A novel probe tester for the characterization of 1 mbit/cm2 bubble memory devices. *Journal of Applied Physics*, 52(3):2392–2394, 1981.

[4] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[5] Leonhard Euler. *Mechanica sive motus scienta analytice exposita... instar supplementi ad Commentar. Acad. scient. imper.* ex typographia Academiae scientarum, 1736.

[6] Leonhard Euler. *Methodus inveniendi lineas curvas maximi minimive proprietate gaudentes (appendix, de curvis elasticis).* Marcum-Michaelem Bousquet, 1744.

[7] Craig G Fraser. Isoperimetric problems in the variational calculus of euler and lagrange. *Historia mathematica*, 19(1):4–23, 1992.

[8] Izrail Moiseevich Gelfand, Sergeĭ Vasilevich Fomin, and Richard A Silverman. *Calculus of variations.* Courier Corporation, 2000.

89

[9] Herbert Goldstein. *Classical mechanics*. Addison Wesley, 2002.

[10] JL Heiberg and Richard Fitzpatrick. Euclid's elements of geometry. 2007.

[11] Shoshichi Kobayashi and Katsumi Nomizu. *Foundations of differential geometry*, volume 1. New York, 1963.

[12] Cornelius Lanczos. *The variational principles of mechanics*, volume 4. Courier Corporation, 1970.

[13] LD Landau and EM Lifshitz. Mechanics, vol. 1. *Course of theoretical physics*, 1976.

[14] Walter Ritz. Über eine neue methode zur lösung gewisser variationsprobleme der mathematischen physik. 1909.

[15] Dirk J Struik. Outline of a history of differential geometry (ii). *Isis*, 20(1):161–191, 1933.

[16] Dirk Jan Struik. Outline of a history of differential geometry: I. *Isis*, 19(1):92–120, 1933.

[17] Bruce van Brunt. The calculus of variations. 2004.

[18] Robert Weinstock. *Calculus of variations: with applications to physics and engineering*. Courier Corporation, 1952.

[19] R.J. Whitinger. Arbitrary drive for magnetic field waveform control, August 23 1983. US Patent 4,400,809.

APPENDICES

A   How to reproduce results

Researchers wishing to reproduce the results we have presented, or wishing to apply the CVA algorithm to their own research, will be pleased to know that we have implemented the algorithm as a multi-platform importable python library and have released this code as LGPLv2.1+ licensed open source software. The library is initially available through the Python Package Index (PyPI) as well as other downstream repositories.

A.1   Installation

Installation of the library directly from PyPI is done with this command:

pip install cva

See https://pypi.python.org/ for a description of how to set up the **pip** installer.

A.2   Quickstart Tutorial

With the library installed, we give here a few examples of its use. In keeping with the tradition of beginning with the smallest example, we provide this two liner:

Hello World example:

```
import cva
cva.examples.case01_torus()
```

You should now see the graphic we show in Figure 4.5.1.

The cva library includes a variety of prepackaged models and metrics. These may

be accessed like this:

Using prepackaged models and metrics:

```python
import cva

cva.solve.select(cva.model.hyperboloid, cva.metric.distance)
sa = (0.0, 0.2) # starting point in <u,v> parameterization space
sb = (1.0, 0.6) # ending point
path = cva.solve.run(sa,sb)
cva.view.draw(path)
```

It is also straightforward to implement your own custom models and metrics. Here

we show an example of a custom model:

Using custom models and metrics:

```python
import cva
import numpy as np

def mymodel(s):
    s,x = cva.model.startmodel(s)

    # start of model mapping
    u = s[:,0]
    v = s[:,1]
    theta = 2.0*np.pi*(0.5-v)
    x[:,0] = (1.0-u)*np.cos(theta)
    x[:,1] = (1.0-u)*np.sin(theta)
    x[:,2] = u
    # end of model mapping

    x = cva.model.finishmodel(x)
    return x

cva.solve.select(mymodel, cva.metric.distance)
sa = (0.0,0.0)
sb = (0.6,0.5)
steps = 4
```

```
path = cva.solve.run(sa,sb,steps)
cva.view.draw(path,title="My Custom Model")
```

When running this code we have these results:

```
step 1: path_integral = 1.613512 after 0.487 seconds
step 2: path_integral = 1.784420 after 1.916 seconds
step 3: path_integral = 1.806089 after 5.246 seconds
step 4: path_integral = 1.811742 after 12.353 seconds
```

and the following graphic:
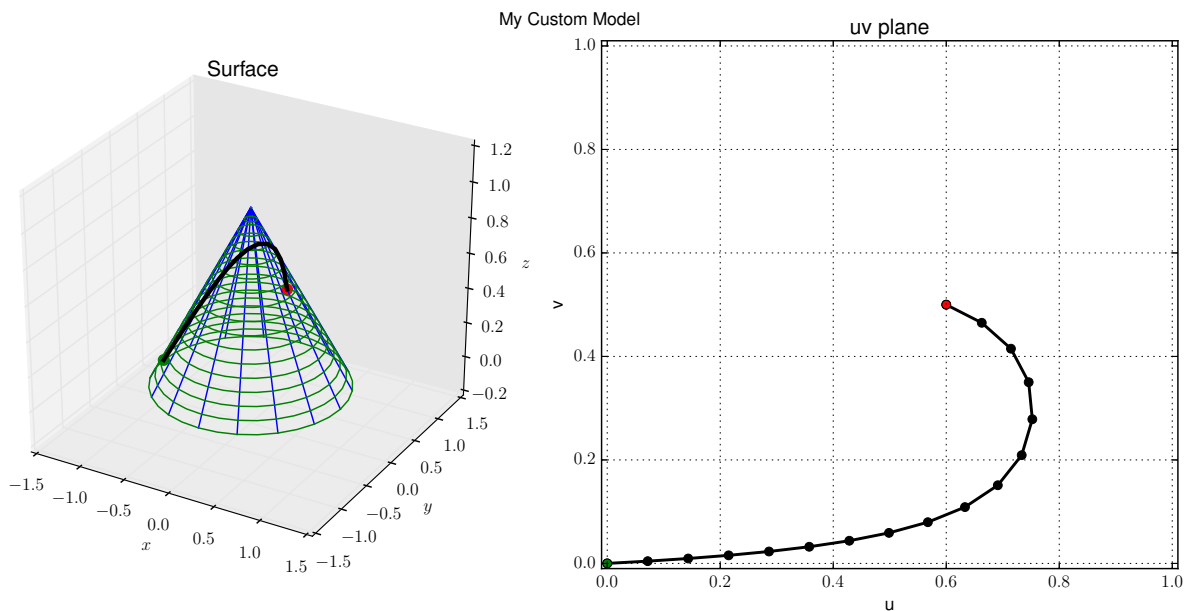
Figure 41: Creating a Custom Model

B Code listings

B.1 solve.py

```python
# -*- coding: utf-8 -*-
#------------------------------------------------------------------------------
#  Copyright (c) 2016, Robert Whitinger <cva_account@wmkt.com>
#
#  Distributed under the terms of the LGPL license either version 2.1 or
#  (at your option) any later version.
#
#  The full license is in the file LICENSE.txt, distributed with this
#     software,
#  and is also available at <http://www.gnu.org/licenses/>
#------------------------------------------------------------------------------
"""
cva.solve

A calculus of variations solver.

This module implements the CVA algorithm for the numerical computation of
minimum paths.
"""

from __future__ import division

# system libraries:
import sys
import numpy as np
import math
from copy import deepcopy
import time
import itertools

# project library
import cva

# Constants
PI = math.pi
TPI = 2.0*PI
EPS = sys.float_info.epsilon
```

```python
# here we store this module's parameter set and make it accessible

# initialize default tuning parameters
_parms = {'tp_starting_trial_space_radius': 0.5,
          'tp_eps_multiplier': 100,
          'tp_straddles': 32,
          'tp_max_trials': 20,
          'trace_minpoint': False,
          'trace_straddle': False,
          'trace_refine': False
          }
def set_parm(parm_name, parm_value):
    """
    cva.solve.set_parm(parm_name, parm_value)
    """
    _parms[parm_name] = parm_value


def get_parm(parm_name):
    """
    parm_value = cva.solve.get_parm(parm_name)
    """
    return _parms[parm_name]


def list_parms():
    """
    keys = cva.solve.list_parms()
    """
    keys = _parms.keys()
    return keys


def select(function_model, function_metric):
    """
    cva.solve.select(function_model, function_metric)

    Parameters
    ----------
    function_model : function
        This parameter assigns a model.
    function_metric : function
        This parameter assigns a metric.

    Returns
    -------
```

```
    nothing

    Notes
    -----
    This function is called prior to cva.solve.run() invokation in order to
    specify a model with an associated metric. See also cva.solve.run().

    Examples
    --------
    import cva
    sa = (0.4,0.5)
    sb = (0.6,0.7)
    cva.solve.select(cva.model.sphere,cva.metric.distance)
    path = cva.solve.run(sa,sb)

    In this example, a spherical model with a Euclidean distance metric
    are specified, and then with this configuration a minimal path is
    calculated and a minimal curve is returned.
    """
    _parms['G'] = function_model
    _parms['M'] = function_metric
    return

# The basic computational element finds a minimum of the objective
    functional
# over a successively smaller trial space.
def minpoint(sa, sb):
    """
    cva.solve.minpoint(sa, sb)

    Parameters
    ----------
    sa : array_like
        This parameter contains a starting point in the parameter space.
    sb : array_like
        This parameter contains an ending point in the parameter space.

    Returns
    -------
    sm : ndarray
        On return sm contains the minpoint between the starting point sa
        and the ending point sb. Given the constraints of the specified
        model and metric.
```

```
Notes
-----
This is the basic building block of the cva algorithm.

Examples
--------
import cva
sa = (0.4,0.5)
sb = (0.6,0.7)
cva.solve.select(cva.model.sphere,cva.metric.distance)
sm = cva.solve.minpoint(sa,sb)
"""
# we determine the dimension of our phase space by examining the sa
    point
sa = np.asarray(sa)
sb = np.asarray(sb)
if np.allclose(sa, sb):
    return sa         # quick return if starting and ending are equal
nparm = np.shape(sa)[0]
# use Gramm-Schmidt to form an n-dimensional space perpendicular to
# the secant line

# step 1: form a non-orthogonal basis space including our secant vector
basis = np.zeros((nparm, nparm))
cartesian_space = np.zeros((nparm, nparm))
for i in range(nparm):
    cartesian_space[i, i] = 1.0
# our non-orthogonal basis must include the primary (secant) unit
    vector
basis[0] = (sb - sa)/np.linalg.norm(sb - sa)
# we select a the set of cartesian unit vectors by eliminating the
    worst choice
worst_choice = np.argmax(np.abs(basis[0]))
# then we list the indices of the best cartesian choices
axis = [axis for axis in range(nparm) if axis != worst_choice]
for i in range(1, nparm):
    # and use them to form the rest of our basis
    basis[i] = cartesian_space[axis[i-1]]
    if np.allclose(basis[i], basis[0]):
        raise ValueError('colinear vectors found in basis')

# step 2: form an orthonormal basis including the secant vector
perp = np.empty((nparm, nparm))
perp[0] = basis[0]       # perp[0] is the secant unit vector
```

97

```python
for i in range(1, nparm): # the Gramm-Schmidt summation
    perp[i] = basis[i]  # starting with the basis vector
    for j in range(i):  # then subtracting previous vector components
        perp[i] -= perp[j]*np.inner(basis[i],
            perp[j])/np.inner(perp[j], perp[j])
    perp[i] = perp[i]/np.linalg.norm(perp[i]) # normalize

# step 3: starting with a midpoint and a radius, find the best path
sm = (sb-sa)/2.0 + sa
radius = np.linalg.norm(sb-sa)*_parms['tp_starting_trial_space_radius']
# our trial space has 5**(n-1) points,
# five points accross each axis centered on the current midpoint
    estimate
trial_shape = []
for i in range(nparm-1):
    trial_shape.append(5)
trial_array_shape = deepcopy(trial_shape)
trial_array_shape.append(nparm)
# trial_space = np.zeros(trial_array_shape) # the set of trial points
# trial_integral = np.zeros(trial_shape) # path summations for each
    trial point
maxtries = _parms['tp_max_trials']
while maxtries > 0 and radius > EPS*_parms['tp_eps_multiplier']:
    maxtries -= 1
    best_integral = np.infty
    # create a tuple of all possible trial index permutations
    trial_index_set = itertools.product(range(5), repeat=nparm-1)
    trial_space = np.zeros(trial_array_shape) # the set of trial points
    trial_integral = np.zeros(trial_shape) # path summations for each
        trial point
    for it in trial_index_set: # loop over all trial permutations
        trial_space[it] = sm # our trial point includes the center
        for i in range(nparm-1): # and spans the basis coordinates
            trial_space[it] += radius*perp[i+1]*(it[i]-2)/2.0
            #print np.linalg.norm(radius*perp[i+1]*(it[i]-2)/2.0)
        trial_integral[it] = _parms['M'](sa, trial_space[it]) +
            _parms['M'](trial_space[it], sb)
        if trial_integral[it] < best_integral:
            best_integral = trial_integral[it]
            bestit = it
    # prepare new trial (center and radius) based on our best path
    if best_integral < np.infty:
        sm = trial_space[bestit]
        radius = radius/2
```

```python
        else:
            print "no solution found"
        # tracing brings a performance penalty, it is turned off by default
        if _parms['trace_minpoint']:
            try:
                _parms['log_minpoint']
            except:
                _parms['log_minpoint'] = []
            log = deepcopy((sm, trial_space, trial_integral, radius))
            _parms['log_minpoint'].append(log)

    return sm


# utility procedure
def _strip_s(s, step):
    # strip uncalculated values from s
    i = 0
    s_temp = np.zeros((2**step+1, np.shape(s)[1]))
    for i in range(2**step+1):
        j = i*2**(_parms['steps']-step)
        s_temp[i] = s[j]
    return s_temp

def path_integral(s, step=False):
    """
    cva.solve.path_integral(s, step=False)

    Parameters
    ----------
    s : array_like
        This parameter specifies a path which is not necessarily minimal.
    step : int (optional)
        This parameter specifies the scope of the straddle operation. The
        default False results in a path integral computation over all of s.

    Returns
    -------
    result : ndarray
        A value corresponding to the piecewise path summation of the curve
        defined in s.

    Notes
    -----
```

```
    none

    """
    if step:
        s = _strip_s(s, step)
    result = 0.0
    for i in range(s.shape[0]-1):
        result += _parms['M'](s[i], s[i+1])
    return result

def straddle(s, step):
    """
    cva.solve.straddle(s,step)

    An internal function called by cva.solve.run()

    Parameters
    ----------
    s : array_like
        This parameter specifies a path which is not necessarily minimal.
    step : int
        This parameter specifies the scope of the straddle operation.

    Returns
    -------
    s : ndarray
        On return s specifies a minimal path of 2**steps + 1 points.

    Notes
    -----
    The straddle operation uses minpoints to build convergent piecewise
        minimal
    paths. This function is used primarily by cva.solve.run(), the primary
    entry point. See cva.solve.run() for example usage.
    """
    try:
        N = _parms['N']
    except:
        N = len(s)-1
    incr = int(N/(2**step))
    for iteration in range(_parms['tp_straddles']):
        for k in range(1, int(N/incr), 2):
            a = (k-1)*incr
            b = (k+1)*incr
```

```python
            s[int(k*N/(2**step))] = minpoint(s[a], s[b])
        for k in range(2, int(N/incr), 2):
            a = (k-1)*incr
            b = (k+1)*incr
            s[int(k*N/(2**step))] = minpoint(s[a], s[b])
        if _parms['trace_straddle']:
            try:
                _parms['log_straddle']
            except:
                _parms['log_straddle'] = []
            log = deepcopy((iter, s ))
            _parms['log_straddle'].append(log)
    return (s)

def refine(s, steps, silent):
    """
    cva.solve.refine(s, steps, silent)

    An internal function called by cva.solve.run()

    Parameters
    ----------
    s : array_like
        This parameter specifies a path.
    steps : int
        This parameter specifies the number of refinement operations to
        be performed. Each refinement approximately doubles the number
        of points in the path.

    Returns
    -------
    s : ndarray
        On return s specifies a minimal path of 2**steps + 1 points.

    Notes
    -----
    The refinement operation uses straddles and minpoints to build
    convergent piecewise minimal paths of increasing length. This function
    is used primarily by cva.solve.run(), the primary entry point.
    See cva.solve.run() for example usage.
    """
    for step in range(1, steps+1):
        s = straddle(s, step)
        run_time = time.time() - _parms['start_time']
```

```python
            if _parms['trace_refine']:
                try:
                    _parms['log_refine']
                except:
                    _parms['log_refine'] = []
                log = deepcopy((step, run_time, s ))
                _parms['log_refine'].append(log)
            if not silent:
                print "step %d: path_integral = %.6f after %.3f seconds" % \
                    (step, path_integral(s, step), run_time)
        return s


def _choose_path(sa, sb):
    # for now we assume that the last parameter is periodic
    N = _parms['N']
    nparm = _parms['nparm']
    s = np.zeros((N+1, nparm))
    s[0] = sa
    s[N] = sb
    # we give the model a chance to declare its periodic axes

    # cause the model to report periodicity
    _parms['G'](sa)
    periodicity = cva.model.get_parm('periodicity')

    # check for a tuple of booleans
    # for now we only implement periodicity in the last parameter
    if periodicity[-1] == True:
        sm = (s[N]-s[0])/2.0 + s[0]
        dist = _parms['M'](sm, s[N])
        if s[0, nparm-1] < s[N, nparm-1]:
            sa_ext = deepcopy(s[0])
            sa_ext[nparm-1] += 1.0
            sm_ext = (s[N]-sa_ext)/2.0 + sa_ext
            dist_ext = _parms['M'](sm_ext, s[N])
            if dist_ext + EPS < dist:
                s[0, nparm-1] += 1.0
        else:
            sb_ext = deepcopy(s[N])
            sb_ext[nparm-1] += 1.0
            sm_ext = (sb_ext-s[0])/2.0 + s[0]
            dist_ext = _parms['M'](sm_ext, s[N])
            if dist_ext + EPS < dist:
                s[N, nparm-1] += 1.0
```

```
        return s

def run(sa, sb, steps=5, silent=False):
    """
    cva.solve.run(sa, sb, steps=5, silent=False)

    Parameters
    ----------
    sa : array_like
        This parameter contains a starting point in the parameter space.
    sb : array_like
        This parameter contains an ending point in the parameter space.
    steps : int, optional
        This parameter specifies the number of refinement steps to be
        executed. Each step approximately doubles the number of points
        in the returned path.
    silent : bool, optional
        In the normal case, cva.solve.run() prints a runtime status
        report. Setting silent = True will disable this reporting.

    Returns
    -------
    s : ndarray
        On return s contains a sequence of points in parameter space which
        define a minimal path for the given starting/ending points, metric,
        and model. The length of s will be 2**(steps) + 1 points.

    Notes
    -----
    This is the primary entry point for the cva.solve module. See also
    cva.solve.select().


    Examples
    --------
    import cva
    sa = (0.4,0.5)
    sb = (0.6,0.7)
    cva.solve.select(cva.model.sphere,cva.metric.distance)
    path = cva.solve.run(sa,sb)

    In this example, the parameterization points sa and sb are mapped into
    their corresponding points on the surface of a unit sphere, and a
    33 point minimal curve is returned.
```

```python
    """
    # we deduce the required parameters from the user's input
    # the length of the starting point sa is the dimension of our phase
        space
    N = 2**steps           # length of path array is N+1
    nparm = len(sa)
    set_parm('start_time', time.time())
    set_parm('sa', np.asarray(sa))
    set_parm('sb', np.asarray(sb))
    set_parm('steps', steps)
    set_parm('N', N)
    set_parm('nparm', nparm)
    set_parm('silent', silent)

    # we create a suitably sized path array and set its starting and
        ending points
    s = _choose_path(sa, sb)
    s = refine(s, steps, silent)
    return s


if __name__ == "__main__":

    print "running cva/solve.py"
```

B.2 metric.py

```python
# -*- coding: utf-8 -*-
#-----------------------------------------------------------------------------
#  Copyright (c) 2016, Robert Whitinger <cva_account@wmkt.com>
#
#  Distributed under the terms of the LGPL license either version 2.1 or
#  (at your option) any later version.
#
#  The full license is in the file LICENSE.txt, distributed with this
#     software,
#  and is also available at <http://www.gnu.org/licenses/>
#-----------------------------------------------------------------------------
"""
cva.metric

This module contains implementations of several metrics. These
metrics accept two points in parameter space and return a single value.
That value may be a distance, or in a more general case it may be
an arbitrary objective functional.
"""


from __future__ import division
import numpy as np
import math

# project library
import cva

# Constants
PI = math.pi
TPI = 2.0*PI

def distance(sa, sb):
    """
    cva.metric.distance(sa, sb)

    Parameters
    ----------
    sa : array_like
        This parameter contains a starting point in the parameter space.
    sb : array_like
```

This parameter contains an ending point in the parameter space.

    Returns
    ----
    metric : float64
        The Euclidean distance between the two surface points corresponding
        to sa and sb.

    Notes
    -----
    none

    Examples
    ----
    import cva
    sa = (0.4,0.5)
    sb = (0.6,0.7)
    cva.solve.select(cva.model.sphere,cva.model.distance)
    metric = cva.model.distance(sa,sb)

    In this example, the parameterization points sa and sb are mapped into
    their corresponding points on the surface of a unit sphere, and the
    Euclidean distance between those two points is returned.
    """

    G = cva.solve.get_parm('G')
    xa = np.asarray(G(sa))   # from point
    xb = np.asarray(G(sb))   # to point
    summation = np.sum((xb-xa)**2)
    metric = math.sqrt(summation)
    return metric

# Brachistochrone objective functional
def brachistochrone_earth(s0, s1):
    """
    cva.metric.brachistochrone_earth(sa, sb)

    Parameters
    ----
    sa : array_like
        This parameter contains a starting point in the parameter space.
    sb : array_like
        This parameter contains an ending point in the parameter space.
```

```
    Returns
    -------
    metric : float64
        The time required to move between the two surface points
            corresponding
        to sa and sb on a straight line path in Earth gravity.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4,0.5)
    sb = (0.6,0.7)
    cva.solve.select(cva.model.tilted_plane,cva.model.brachistochrone_earth)
    metric = cva.model.brachistochrone_earth(sa,sb)

    In this example, the parameterization points sa and sb are mapped into
    their corresponding points on the surface of a tilted plane, and the
    time required to traverse between those two points is returned.
    """
    G = cva.solve.get_parm('G')
    sa = cva.solve.get_parm('sa')
    x0, y0, z0 = G(s0[:])[0] # from point
    x1, y1, z1 = G(s1[:])[0] # to point
    _, _, za = G(sa[:])[0] # model starting point
    g = 9.8
    if z1 >= za:
        metric = np.infty # we can't reach points higher than our starting
            point
    else:
        metric =
            math.sqrt(((x1-x0)**2+(y1-y0)**2+(z1-z0)**2)/(2.0*g*(za-z1)))
    return metric

def brachistochrone_moon(s0, s1):
    """
    cva.metric.brachistochrone_moon(sa, sb)

    Parameters
    ----------
    sa : array_like
```

```
        This parameter contains a starting point in the parameter space.
    sb : array_like
        This parameter contains an ending point in the parameter space.

    Returns
    -------
    metric : float64
        The time required to move between the two surface points
            corresponding
        to sa and sb on a straight line path in Moon gravity.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4,0.5)
    sb = (0.6,0.7)
    cva.solve.select(cva.model.tilted_plane,cva.model.brachistochrone_moon)
    metric = cva.model.brachistochrone_moon(sa,sb)

    In this example, the parameterization points sa and sb are mapped into
    their corresponding points on the surface of a tilted plane, and the
    time required to traverse between those two points is returned.
    """
    G = cva.solve.get_parm('G')
    sa = cva.solve.get_parm('sa')
    x0, y0, z0 = G(s0[:])[0] # from point
    x1, y1, z1 = G(s1[:])[0] # to point
    _, _, za = G(sa[:])[0] # model starting point
    g = 1.62
    if z1 >= za:
        metric = np.infty # we can't reach points higher than our starting
            point
    else:
        metric =
            math.sqrt(((x1-x0)**2+(y1-y0)**2+(z1-z0)**2)/(2.0*g*(za-z1)))
    return metric

def schwarzschild(s0, s1):
    """
    cva.metric.schwarzschild(sa, sb)
```

```
Parameters
----------
sa : array_like
    This parameter contains a starting point in the parameter space.
sb : array_like
    This parameter contains an ending point in the parameter space.

Returns
-------
metric : float64
    The distance in spacetime between the two events in curved space,
    near a massive object, corresponding to sa and sb.

Notes
-----
none

Examples
--------
import cva
sa = (0.4,0.5)
sb = (0.6,0.7)
cva.solve.select(cva.model.blackhole,cva.model.schwarzschild)
metric = cva.model.schwarzschild(sa,sb)

In this example, the parameterization points sa and sb are mapped into
their corresponding events near a supermassive blackhole, and the
spacetime distance between those two events is returned.

The Schwarzschild metric forms the basis of general relativity.
"""
G = cva.solve.get_parm('G')
if np.ndim(s0) == 1:
    nparm = len(s0)
else:
    nparm = np.shape(s0)[1]
if nparm == 2:          # model surface at fixed time
    x0, y0, z0 = G(s0[:])[0] # from point
    x1, y1, z1 = G(s1[:])[0] # to point
    ct0 = ct1 = 0.0
else:
    raise NotImplementedError
C = 299792458.0 # speed of light (m/s)
```

```python
        Ms = 1.98855e+30 # mass of the Sun (kg)
        Mb = 4.31e+6 * Ms # mass of Sagitarius A* (kg)
        Gc = 6.67384e-11 # gravitational constant (m^3 kg^-1 s^-2)
        gm = Gc*Mb
        Rs = 2.0*gm/(C*C) # Schwarzschild radius
        r0 = math.sqrt((x0-0.5)**2 + (y0-0.5)**2 + (z0-0.5)**2)
        r1 = math.sqrt((x1-0.5)**2 + (y1-0.5)**2 + (z1-0.5)**2)
        r = (r1+r0)/2.0
        theta0 = theta1 = theta = PI/2.0
        phi0 = (PI/2.0)-np.arctan2(z0, y0)
        if phi0 > PI:
            phi0 = phi0 - TPI
        phi1 = (PI/2.0)-np.arctan2(z1, y1)
        if phi1 > PI:
            phi1 = phi1 - TPI
        try:
            dphi = phi1 - phi0
            if dphi > PI:
                dphi = TPI - dphi
            metric = (1/C)*math.sqrt(-(1-(Rs/r))*(ct1-ct0)**2 +
                (1/(1-(Rs/r)))*(r1-r0)**2 + r**2 * (theta1-theta0)**2 + r**2 *
                np.sin(theta)**2 * (dphi**2))
        except:
            metric = np.infty
        return metric


def minkowski(s0, s1):
    """
    cva.metric.minkowski(sa, sb)

    Parameters
    ----------
    sa : array_like
        This parameter contains a starting point in the parameter space.
    sb : array_like
        This parameter contains an ending point in the parameter space.

    Returns
    -------
    metric : float64
        The distance in spacetime between the two events in flat space
        (a region of space void of massive objects) corresponding to sa and
            sb.
```

```
Notes
-----
none

Examples
--------
import cva
sa = (0.4,0.5)
sb = (0.6,0.7)
cva.solve.select(cva.model.inflating_sphere,cva.model.minkowski)
metric = cva.model.minkowski(sa,sb)

In this example, the parameterization points sa and sb are mapped into
their corresponding events in spacetime, and the spacetime distance
between those two events is returned.

The Minkowski metric forms the basis of special relativity.
"""
G = cva.solve.get_parm('G')
x0, y0, z0, ct0 = G(s0[:])[0] # from point
x1, y1, z1, ct1 = G(s1[:])[0] # to point
metric = math.sqrt(abs((ct1-ct0)**2 -(x1-x0)**2 - (y1-y0)**2 -
    (z1-z0)**2))
if ct1 < ct0:
    metric = np.infty # we don't allow negative time movements
return metric


if __name__ == "__main__":

    print "running cva/metric.py"
```

B.3 model.py

```python
# -*- coding: utf-8 -*-
#-----------------------------------------------------------------------------
#  Copyright (c) 2016, Robert Whitinger <cva_account@wmkt.com>
#
#  Distributed under the terms of the LGPL license either version 2.1 or
#  (at your option) any later version.
#
#  The full license is in the file LICENSE.txt, distributed with this
#    software,
#  and is also available at <http://www.gnu.org/licenses/>
#-----------------------------------------------------------------------------
"""
cva.model

This module contains implementations of several surface models. These
implementations are in the form of mappings from a <u,v> plane to an
    <x,y,z>
surface, or in the general case of higher dimensional hypersurfaces in
the form of mappings from a <u0,u1,...uj> parameter space into an
<x0,x1,...,xk> hypersurface.

We adopt the convention that the surface is described by u-parameters in
the range of [0,1]. Models are required to accept out of range inputs
over the range of [-1,2] and return valid manifold surface mappings. In
    other
words, the responsibility for wraparound lies with the model. This is
reasonable since wraparound rules tend to be model-specific.
"""

from __future__ import division
import numpy as np
import math
from copy import deepcopy

# Constants
PI = math.pi
TPI = 2.0*PI

# here we store this module's parameter set and make it accessible
global _parms
# initialize default parameters
```

112

```python
_parms = {'periodicity' : (False, True),
          'sdim' : 2,
          'xdim' : 3
          }
def set_parm(parm_name, parm_value):
    """
    cva.model.set_parm(parm_name, parm_value)
    """
    _parms[parm_name] = parm_value


def get_parm(parm_name):
    """
    parm_value = cva.model.get_parm(parm_name)
    """
    return _parms[parm_name]


def list_parms():
    """
    keys = cva.solve.list_parms()
    """
    keys = _parms.keys()
    return keys


def startmodel(s, periodicity=True, xdim = 3):
    """
    startmodel(s)

    Model prologue.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length, nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1, nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].
    periodicity : tuple of boolean (optional)
        By default, models are assumed to be periodic in all dimensions.
        When creating a model that departs from this default, then
        the periodicity of each axis can be defined. For example a
        model with a 2-dimensional parameter space where neither axis
        is periodic would be specified with periodicity=(False,False).
    xdim : integer (optional)
```

By default models are assumed to map into 3-dimensional space.
            When creating a model with mapping into a higher dimensional space
            then the dimension is specified with the xdim parameter.

    Returns
    ----
    s : ndarray of parameter space points
            The s parameter is returned in the form of a row vector.
    x : ndarray of surface points
            In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
            and x[:,2] to the z. The zero value array x has a dimension
            of [len(s), xdim].

    Notes

    This function is called by every model immediately on entry. See also
    finishmodel().
    """
    global entry_id
    entry_id = id(s)
    set_parm('xdim', xdim)
    # if we are called with an array we just use it after making a local
        copy
    if type(s) is np.ndarray:
        local_copy = deepcopy(s)
    else:
        # but we also accept tuples and lists, others generate an error
        if type(s) is not tuple and type(s) is not list:
            raise TypeError("Expected tuple or list but got %s" % (type(s)))
        local_copy = np.asarray(s)
    # single points are converted into a uniform array format
    if np.ndim(local_copy) == 1:
        local_copy = local_copy.reshape(1, -1)
    s = local_copy
    # create a suitably sized array x for the model's path array
    length, nparameters = np.shape(s)
    set_parm('sdim', nparameters)
    x = np.zeros((length, xdim))

    # save periodicity
    # periodicity = True means that this model takes the default, that is
    # the last parameter in our list is periodic.
    # A model may pass a mask in order to override, for example,
    # a model with two parameters may set periodicity=(False,False) in

```python
        # order to declare that neither parameter is periodic.

        # if the model does not override then we apply the default which
        # is to assume that the last parameter is periodic
        if periodicity == True:
            periodicity = (False,)*(np.shape(s)[1]-1)
            periodicity += (True,)
        set_parm('periodicity', periodicity)
        exit_id = (id(s))
        # make sure we have a local copy of s for our model
        assert entry_id != exit_id
        return s, x

def finishmodel(x, s):
    """
    finishmodel(x, s)

    Model epilogue.

    Parameters
    ----------
    x : ndarray of surface points
    s : ndarray of parameter space points

    Returns
    -------
    x : ndarray of surface points

    Notes
    -----
    This function is called by every model just prior to its return. See
    also startmodel().
    """
    # code any general model exit conventions here
    # verify that the model did not alter s
    exit_id = id(s)
    assert entry_id != exit_id
    return x

def vertical_plane(s):
    """
    Model a vertical plane in R3.

    Parameters
```

```
----------
s : array_like
    This parameter contains one or more points in the parameter
    space. Multiple parameters are of the form s[length,nparm] where
    nparm is the dimensionality of the parameter space. Single
    points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
    as a list [u0, u1, ...].

Returns
-------
x : ndarray of surface points
    In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
    and x[:,2] to the z.

Notes
-----
none

Examples
--------
import cva
sa = (0.4, 0.5)
x = cva.model.vertical_plane(sa)

In this example, the point defined by u = 0.4 and v = 0.5 is mapped
    into
its corresponding point on the model's surface and that result is
    returned
as a row array of points in cartesian coordinates.
"""
s, x = startmodel(s, (False, False))

# start of model mapping
u = s[:, 0]
v = s[:, 1]
x[:, 0] = u
x[:, 1] = np.zeros_like(u) # this obtains a zero vector with the shape
    of u
x[:, 2] = v
# end of model mapping

x = finishmodel(x, s)
return x
```

```python
def tilted_plane(s):
    """
    Model a tilted plane in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.tilted_plane(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s, (False, False))

    # start of model mapping
    u = s[:, 0]
    v = s[:, 1]
    alpha = -np.pi/4.0
    # rotate around the x axis by pi/4
    x[:, 0] = u
    x[:, 1] = v*np.cos(alpha)
```

```python
    x[:, 2] = -v*np.sin(alpha)
    # end of model mapping

    x = finishmodel(x, s)
    return x

def hyperboloid(s):
    """
    Model a hyperboloid in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.hyperboloid(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s)

    # start of model mapping
```

```python
    u = s[:, 0]
    v = s[:, 1]
    theta = (v-0.5)*TPI
    x[:, 0] = np.cosh(3.0*(0.5-u))*np.cos(theta)
    x[:, 1] = np.cosh(3.0*(0.5-u))*np.sin(theta)
    x[:, 2] = np.sinh(3.0*(0.5-u))
    # end of model mapping

    x = finishmodel(x, s)
    return x


def cylinder(s):
    """
    Model a cylinder in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.cylinder(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
```

```python
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s)

    # start of model mapping
    u = s[:, 0]
    v = s[:, 1]
    x[:, 0] = np.cos(v*TPI)
    x[:, 1] = np.sin(v*TPI)
    x[:, 2] = 1.0 - u
    # end of model mapping

    x = finishmodel(x, s)
    return x

def capped_cylinder(s):
    """
    Model a capped_cylinder in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.capped_cylinder(sa)
```

```
    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s)

    # start of model mapping
    u = s[:, 0]
    v = s[:, 1]
    p = 4.0/32.0
    theta = (v-0.5)*TPI
    x[:, 0] = np.cos(theta)
    x[:, 0] = np.where(u < p, x[:, 0]*(u/p), x[:, 0])
    x[:, 0] = np.where(u > (1.0-p), x[:, 0]*(1.0-u)/p, x[:, 0])
    x[:, 1] = np.sin(theta)
    x[:, 1] = np.where(u < p, x[:, 1]*(u/p), x[:, 1])
    x[:, 1] = np.where(u > (1.0-p), x[:, 1]*(1.0-u)/p, x[:, 1])
    x[:, 2] = (u-p)/(1.0-(2.0*p))
    x[:, 2] = np.where(u < p, 0.0, x[:, 2]) # add top cap
    x[:, 2] = np.where(u > (1.0-p), 1.0, x[:, 2]) # add bottom cap

    # end of model mapping

    x = finishmodel(x, s)
    return x

def moebius(s):
    """
    Model a Moebius strip in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
```

```
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.moebius(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s)

    # start of model mapping
    band_width = 2.0
    u = s[:, 0] # u runs around the strip
    v = s[:, 1] # v runs from edge to edge
    w = (u-0.5)*(band_width)
    x[:, 0] = (1.0+w*np.cos(v*PI))*np.cos(v*TPI)
    x[:, 1] = (1.0+w*np.cos(v*PI))*np.sin(v*TPI)
    x[:, 2] = w*np.sin(v*PI)
    # end of model mapping

    x = finishmodel(x, s)
    return x

def torus(s):
    """
    Model a torus in R3.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
```

```
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5)
    x = cva.model.torus(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s)

    # start of model mapping
    u = s[:, 0]
    v = s[:, 1]
    R = 2 # the distance from the origin to the center of the torus
    D = 1 # the diameter of the torus
    phi = (u-0.5)*TPI       # where phi in [-pi,pi]
    theta = (v-0.5)*TPI # where theta in [-pi,pi]
    x[:, 0] = (R + D*np.cos(phi))*np.cos(theta)
    x[:, 1] = (R + D*np.cos(phi))*np.sin(theta)
    x[:, 2] = D*np.sin(phi)
    # end of model mapping

    x = finishmodel(x, s)
    return x

def sphylinder(s):
    """
```

```
Model a sphylinder in R3.

Parameters
----------
s : array_like
    This parameter contains one or more points in the parameter
    space. Multiple parameters are of the form s[length,nparm] where
    nparm is the dimensionality of the parameter space. Single
    points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
    as a list [u0, u1, ...].

Returns
-------
x : ndarray of surface points
    In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
    and x[:,2] to the z.

Notes
-----
none

Examples
--------
import cva
sa = (0.4, 0.5)
x = cva.model.vertical_plane(sa)

In this example, the point defined by u = 0.4 and v = 0.5 is mapped
    into
its corresponding point on the model's surface and that result is
    returned
as a row array of points in cartesian coordinates.
"""
s, x = startmodel(s)

# start of model mapping

# "latitude" ranges from u=0 (north pole) to u=1 (south pole) with
    special wrapping
# "longitude ranges from v=0 to v=1 (2pi) with wrapping
s[:, 1] = np.abs(s[:, 1])    # wraparound v assuming negative and
    positive latitudes are equal
s[:, 1] = np.mod(s[:, 1], 2.0) # wraparound v handling 2pi multiples
```

```python
    s[:, 0] = np.where(s[:, 1] > 1.0, 1.0-s[:, 0], s[:, 0]) # wraparound u
        at "south pole"
    s[:, 1] = np.where(s[:, 1] > 1.0, 2.0-s[:, 1], s[:, 1]) # wraparound v
        at "south pole"
    u = s[:, 0]
    v = s[:, 1]
    # Convert the unit u,v plane into phi, theta representation
    phi = u*PI
    theta = (v-0.5)*TPI
    p = 1.0/2.0
    # Convert to rectangular coordinate system
    x[:, 0] = (np.sin(phi)**p)*np.cos(theta)
    x[:, 1] = (np.sin(phi)**p)*np.sin(theta)
    x[:, 2] = np.where(u < 0.50, np.abs(np.cos(phi))**p,
        -np.abs(np.cos(phi))**p)
    # end of model mapping

    x = finishmodel(x, s)
    return x

def latlon(lat, lon):
    """
    A utility function to convert latitude and longitude into a <u,v>
    representation suitable for use in the cva.model.earth.

    Parameters
    ----------
    lat : float
        Latitude in the range of -90 (south pole) to +90 (north pole).
    lon : float
        Longitude in the range of -180 (west) to +180 degrees (east).

    Returns
    -------
    sa : tuple of <u,v> coordinates

    Notes
    -----
    See also cva.model.earth()
    """
    sa = ((90.0 - lat)/180.0, (lon+180.0)/360.0)
    return sa

def earth(s):
```

```
"""
Model the Earth (WGS84) in R3.

Parameters
----------
s : array_like
    This parameter contains one or more points in the parameter
    space. Multiple parameters are of the form s[length,nparm] where
    nparm is the dimensionality of the parameter space. Single
    points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
    as a list [u0, u1, ...].

Returns
-------
x : ndarray of surface points
    In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
    and x[:,2] to the z.

Notes
-----
This is the WGS84 (World Geodetic System 1984) geocentric
equipotential ellipsoid model of the Earth surface as projected onto
    the
ECEF (Earth centered Earth fixed) cartesian coordinate system in units
of kilometers.

Circumference of Earth:

40,075.017 km (equatorial)
40,007.860 km (meridional)


See also the utility function cva.model.latlon().

Examples
--------
import cva
sa = (0.4, 0.5)
x = cva.model.earth(sa)

In this example, the point defined by u = 0.4 and v = 0.5 is mapped
    into
its corresponding point on the model's surface and that result is
    returned
```

126

```python
    as a row array of points in cartesian coordinates.

    import cva
    lat = 36.303181
    lon = -82.368280
    sa = cva.model.latlon(lat,lon)
    x = cva.model.earth(sa)

    In this example we show the use of the utility function
    cva.model.latlon() to find the coordinates of a point on the
    Earth's surface corresponding to a geographical location expressed
    in decimal latitude and longitude.
    """
    s, x = startmodel(s)

    # start of model mapping

    # WGS84 defining constants
    a = 6378.1370 # Semi-major axis in units of kilometers
    f = 1.0/298.257223563 # flattening
    # Derived constants
    # b = a*(1-f) # Semi-minor axis (6356752.31425)
    e2 = 2*f - f*f # First eccentricity squared
    # "latitude" ranges from u=0 (north pole) to u=1 (south pole) with
        special wrapping
    # "longitude ranges from v=0 (-pi) to v=1 (pi) with wrapping
    u = np.empty_like(s[:, 0])
    v = np.empty_like(s[:, 1])
    u = np.where(s[:, 0] > 1.0, 2.0-s[:, 0], s[:, 0]) # if u is wrapped at
        south pole
    v = np.where(s[:, 0] > 1.0, s[:, 1]+0.5, s[:, 1]) # then wrap v
        correspondingly
    u = np.where(s[:, 0] < 0.0, -s[:, 0], s[:, 0]) # if u is wrapped at
        north pole
    v = np.where(s[:, 0] < 0.0, s[:, 1]+0.5, s[:, 1]) # then wrap v
        correspondingly
    # Convert the unit u,v plane into phi, theta representation
    phi = u*PI        # where phi ranges from 0 (north pole) to pi (south
        pole)
    theta = (v-0.5)*TPI # theta in [-pi,pi] where theta = 0 is on the
        prime meridian
    # Convert to ECEF rectangular coordinate system
    Nphi = a/(np.sqrt(1+e2*np.cos(phi)**2))
    x[:, 0] = Nphi*np.sin(phi)*np.cos(theta)
```

```python
    x[:, 1] = Nphi*np.sin(phi)*np.sin(theta)
    x[:, 2] = Nphi*np.cos(phi)
    # end of model mapping

    x = finishmodel(x, s)
    return x


def blackhole(s):
    """
    Model a region of space as a plane intersecting a massive object.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        where x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.0, 0.4, 0.5)
    x = cva.model.blackhole(sa)

    In this example, the point defined by u = 0.4 and v = 0.5 is mapped
        into
    its corresponding point on the model's surface and that result is
        returned
    as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s, periodicity=(False, False)) # we declare that
        neither axis is periodic
```

```python
    # start of model mapping

    u = s[:, 0]
    v = s[:, 1]
    C = 299792458.0 # speed of light (m/s)
    Ms = 1.98855e+30 # mass of the Sun (kg)
    Mb = 4.31e+6 * Ms # mass of Sagitarius A* (kg)
    Gc = 6.67384e-11 # gravitational constant (m^3 kg^-1 s^-2)
    gm = Gc * Mb      # we use the mass of the Sgr A* supermassive
        blackhole
    Rs = 2.0*gm/(C*C) # Schwarzschild radius
    R = 20.0 * Rs     # our region is 10x the Schwarzschild radius

    x[:, 0] = u * 0.0
    x[:, 1] = (u - 0.5) * R
    x[:, 2] = (v - 0.5) * R
    # end of model mapping

    x = finishmodel(x, s)
    return x

def inflating_sphere(s):
    """
    Model a sphere inflating in spacetime at 1/2 the speed of light.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none
```

```
Examples
--------
import cva
sa = (0.4, 0.5, 0.6)
x = cva.model.inflating_sphere(sa)

In this example, the point defined by ct = 0.4, u = 0.5, and v = 0.6 is
mapped into its corresponding point on the model's surface and that
    result
is returned as a row array of points in cartesian coordinates.
"""
s, x = startmodel(s, xdim=4)

# start of model mapping

# u[0] = ct (time * light speed)
# u[1] = "latitude" ranges from u[1]=0 (north pole) to u[1]=1 (south
    pole) with special wrapping
# u[2] = "longitude ranges from v=0 (-pi) to v=1 (pi) (periodic)
ct = s[:, 0]
u = np.empty_like(s[:, 1])
v = np.empty_like(s[:, 2])
u = np.where(s[:, 1] > 1.0, 2.0-s[:, 1], s[:, 1]) # if u is wrapped at
    south pole
v = np.where(s[:, 1] > 1.0, s[:, 2]+0.5, s[:, 2]) # then wrap v
    correspondingly
u = np.where(s[:, 1] < 0.0, -s[:, 1], s[:, 1]) # if u is wrapped at
    north pole
v = np.where(s[:, 1] < 0.0, s[:, 2]+0.5, s[:, 2]) # then wrap v
    correspondingly
# Convert the unit u,v plane into phi, theta representation
phi = u*PI         # where phi ranges from 0 (north pole) to pi (south
    pole)
theta = (v-0.5)*TPI # theta in [-pi,pi] where theta = 0 is on the
    prime meridian

# define the rate of inflation
radius = 0.5*ct+0.5 # inflation rate is 1/2 light speed
# Convert to rectangular coordinate system
x[:, 0] = np.sin(phi)*np.cos(theta)*radius
x[:, 1] = np.sin(phi)*np.sin(theta)*radius
x[:, 2] = np.cos(phi)*radius
x[:, 3] = ct
```

```python
    # end of model mapping

    x = finishmodel(x, s)
    return x

def collapsing_sphere(s):
    """
    Model a sphere collapsing in spacetime at 1/2 the speed of light.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].

    Returns
    -------
    x : ndarray of surface points
        In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
        and x[:,2] to the z.

    Notes
    -----
    none

    Examples
    --------
    import cva
    sa = (0.4, 0.5, 0.6)
    x = cva.model.collapsing_sphere(sa)

    In this example, the point defined by ct = 0.4, u = 0.5, and v = 0.6 is
    mapped into its corresponding point on the model's surface and that
        result
    is returned as a row array of points in cartesian coordinates.
    """
    s, x = startmodel(s, xdim=4)

    # start of model mapping

    # u[0] = ct (time * light speed)
```

```python
        # u[1] = "latitude" ranges from u[1]=0 (north pole) to u[1]=1 (south
            pole) with special wrapping
        # u[2] = "longitude ranges from v=0 (-pi) to v=1 (pi) (periodic)
        ct = s[:, 0]
        u = np.empty_like(s[:, 1])
        v = np.empty_like(s[:, 2])
        u = np.where(s[:, 1] > 1.0, 2.0-s[:, 1], s[:, 1]) # if u is wrapped at
            south pole
        v = np.where(s[:, 1] > 1.0, s[:, 2]+0.5, s[:, 2]) # then wrap v
            correspondingly
        u = np.where(s[:, 1] < 0.0, -s[:, 1], s[:, 1]) # if u is wrapped at
            north pole
        v = np.where(s[:, 1] < 0.0, s[:, 2]+0.5, s[:, 2]) # then wrap v
            correspondingly
        # Convert the unit u,v plane into phi, theta representation
        phi = u*PI         # where phi ranges from 0 (north pole) to pi (south
            pole)
        theta = (v-0.5)*TPI # theta in [-pi,pi] where theta = 0 is on the
            prime meridian

        # define the rate of deflation
        radius = 1.0-0.5*ct # deflation rate is 1/2 light speed
        # Convert to rectangular coordinate system
        x[:, 0] = np.sin(phi)*np.cos(theta)*radius
        x[:, 1] = np.sin(phi)*np.sin(theta)*radius
        x[:, 2] = np.cos(phi)*radius
        x[:, 3] = ct
        # end of model mapping

        x = finishmodel(x, s)
        return x

def sphere(s):
    """
    Model a sphere in R^N.

    Parameters
    ----------
    s : array_like
        This parameter contains one or more points in the parameter
        space. Multiple parameters are of the form s[length,nparm] where
        nparm is the dimensionality of the parameter space. Single
        points are accepted as an ndarray s[1,nparm], tuple (u0,u1,...), or
        as a list [u0, u1, ...].
```

```
Returns
-------
x : ndarray of surface points
    In R^3, x[:,0] corresponds to the x dimension, x[:,1] to the y
    and x[:,2] to the z.

Notes
-----
This function generalizes an n-dimensional sphere by examining its
input s to determine the number of parameters in the request. For
    example
a parameter request of dimension 2 (s[0]=latitude, s[1=longitude)
    implies
a 2-sphere in R^3. A three parameter request (s[0]=first latitude,
s[1]=second latitude, s[2]=longitude implies a 3-sphere in R^4, etc.

Examples
--------
import cva
sa = (0.4, 0.5)
x,y,z = cva.model.sphere(sa)

In this example, the point defined by u = 0.4 and v = 0.5 is mapped
    into
its corresponding point on the model's surface and that result is
    returned
as a row array of points in cartesian coordinates.
"""
s, x = startmodel(s)
sdim = get_parm('sdim')
s, x = startmodel(s, xdim=sdim + 1)

# start of model mapping
n = np.shape(s)[1]

# Convert to rectangular coordinate system
prod = 1.0
for i in range(n-1):
    x[:, n-i] = prod * np.cos(np.pi * s[:, i])
    prod = prod * np.sin(np.pi * s[:, i])
x[:, 0] = prod * np.cos(2*np.pi * (s[:, n-1]-0.5))
x[:, 1] = prod * np.sin(2*np.pi * (s[:, n-1]-0.5))
# end of model mapping
```

```python
    x = finishmodel(x, s)
    return x


if __name__ == "__main__":

    print "running cva/model.py"
```

B.4 view.py

```python
# -*- coding: utf-8 -*-
#-----------------------------------------------------------------------------
#  Copyright (c) 2016, Robert Whitinger <cva_account@wmkt.com>
#
#  Distributed under the terms of the LGPL license either version 2.1 or
#  (at your option) any later version.
#
#  The full license is in the file LICENSE.txt, distributed with this
#    software,
#  and is also available at <http://www.gnu.org/licenses/>
#-----------------------------------------------------------------------------
"""
cva.view

A matplotlib wrapper for creating graphical representations of cva
    solutions.
"""

# system libraries:
from __future__ import division
import numpy as np
import matplotlib as mpl
#matplotlib.use('qt4agg')
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import warnings
warnings.filterwarnings("ignore", category=DeprecationWarning)
import matplotlib.cbook
warnings.filterwarnings("ignore", category=matplotlib.cbook.mplDeprecation)
import itertools

# project libraries:
import cva
from cva import model

global initialized
initialized = False

# here we store this module's parameter set and make it accessible
# initialize default parameters if any
_parms = {
```

```python
    }
def set_parm(parm_name, parm_value):
    """
    cva.view.set_parm(parm_name, parm_value)
    """
    _parms[parm_name] = parm_value


def get_parm(parm_name):
    """
    parm_value = cva.view.get_parm(parm_name)
    """
    return _parms[parm_name]


def list_parms():
    """
    keys = cva.view.list_parms()
    """
    keys = _parms.keys()
    return keys


def _init(title="CVA Solution", **kwargs):
    # To display an n-dimensional surface using 3-dimensional views,
    # we need to show combinations of n taken n-3 at a time. For
    # example a 3-d object is represented in one 3-d view, a 4-d
    # object in four 3-d views, a 5-d object in 10 3-d views, etc.
    #
    global G, P, sdim, xdim, nxviews, nsviews, surface_index, parm_index,
        surface_axes, parm_axes
    global fig, ax

    G = cva.solve.get_parm('G')
    P = cva.solve.get_parm('steps')
    sdim = cva.model.get_parm('sdim')
    xdim = cva.model.get_parm('xdim')

#    plt.ion()
    plt.rc('text', usetex=True)
    mpl.rcParams['legend.fontsize'] = 20

    # Calculate the number of views needed
    surface_axes = [x for x in range(xdim)]
    parm_axes = [x for x in range(sdim)]
    if xdim == 3:
        surface_index = [(0,)]
```

```python
    else:
        surface_index = list(itertools.combinations(surface_axes, xdim-3))
    nxviews = len(surface_index)
    if sdim <= 3:
        nsviews = 1    # setting the number of parameter views
        parm_index = [(surface_index[-1][0] +1,)]
    else:
        parm_index = list(itertools.combinations(parm_axes, sdim-3))
        nsviews = len(parm_index)

    nviews = nxviews + nsviews

    fig = plt.figure(figsize=(12, int((6*nviews+1)/2)), dpi=72)
    fig.suptitle(title)
    ax = [x for x in range(nviews)]
    if xdim == 3:
        view = 0
        ax[view] = fig.add_subplot(1, 2, 1, projection='3d')
    else:
        for view in range(nxviews):
            ax[view] = fig.add_subplot(int((len(ax)+1)/2), 2, view+1,
                projection='3d')
    view += 1
    if sdim == 2:
        ax[view] = fig.add_subplot(int((len(ax)+1)/2), 2, view+1)
        ax[view].set_xlim(0.0, 1.0)
        ax[view].set_ylim(0.0, 1.0)
        ax[view].grid(True)
    else:
        for view in range(nxviews, nxviews + nsviews):
            ax[view] = fig.add_subplot(int((len(ax)+1)/2), 2, view+1,
                projection='3d')
            ax[view].set_xlim(0, 1)
            ax[view].set_ylim(0, 1)
            ax[view].set_zlim(0, 1)
    return


def _draw_manifold(title=False, **kwargs):
    # view a hypersurface in n-dimensions
    #
    N = 33
    decimation = 8
    s = np.zeros((N, xdim-1))
```

```python
grid = np.linspace(0.0, 1.0, N)
colors = ["b", "g", "r", "m"]

for view in range(len(surface_index)):
    if xdim == 3:
        axis = (0,1,2)
        decimation = 2
        if title == False:
            title = "Surface"
        ax[view].set_title(title)
        ax[view].set_xlabel(r"$x$")
        ax[view].set_ylabel(r"$y$")
        ax[view].set_zlabel(r"$z$")
    else:
        axis = [axis for axis in surface_axes if axis not in
            surface_index[view]]
        if title == False:
            title = "Hypersurface"
        strng = "x_%d" % (surface_index[view][0])
        for i in range(1, len(surface_index[view])):
            strng += ", x_%d" % (surface_index[view][i])
        ax[view].set_title(r"%s (slice at $%s=0$)" % (title, strng))
        ax[view].set_xlabel(r"$x_%d$" % (axis[0]))
        ax[view].set_ylabel(r"$x_%d$" % (axis[1]))
        ax[view].set_zlabel(r"$x_%d$" % (axis[2]))

    # for this view we draw each axis that is not set to zero
    for i in parm_axes:
        # for each parameter axis, i, we fix the remaining axes on a
            grid
        # and then plot the primary axis
        span = [span for span in parm_axes if span != i]
        # get a list of all grid intersections for the fixed parameters
        points = list(itertools.product(grid[::decimation],
            repeat=len(span)))
        # for each grid point, we span the primary parameter and plot
            the resulting line
        for point in points:
            for j in range(N):
                s[j, i] = grid[j]
                for k in range(len(span)):
                    s[j, span[k]] = point[k]
            # we have a line in s ready to plot
            x = G(s)
```

```python
                ax[view].plot(x[:, axis[0]], x[:, axis[1]], x[:, axis[2]],
                    "-", color=colors[i%len(colors)], linewidth=1,
                    linestyle="-")
            try:
                ax[view].set_xlim(cva.view.get_parm('xyzlim'))
                ax[view].set_ylim(cva.view.get_parm('xyzlim'))
                ax[view].set_zlim(cva.view.get_parm('xyzlim'))
            except:
                pass
            try:
                ax[view].set_xlim(cva.view.get_parm('xlim'))
            except:
                pass
            try:
                ax[view].set_ylim(cva.view.get_parm('ylim'))
            except:
                pass
            try:
                ax[view].set_zlim(cva.view.get_parm('zlim'))
            except:
                pass

    for parm_view in range(len(parm_index)):
        view = parm_view+len(surface_index)
        if sdim == 2:
            ax[view].set_title("uv plane")
            ax[view].set_xlabel("u")
            ax[view].set_ylabel("v")
        if sdim == 3:
            ax[view].set_title("uvw view")
            ax[view].set_xlabel("u")
            ax[view].set_ylabel("v")
            ax[view].set_zlabel("w")
        elif sdim > 3:
            axis = [axis for axis in parm_axes if axis not in
                parm_index[parm_view]]
            str = "u_%d" % (parm_index[parm_view][0])
            for i in range(1, len(parm_index[parm_view])):
                str += ", u_%d" % (parm_index[parm_view][i])
            title = "Phase Space View"
            ax[view].set_title(r"%s (slice at $%s=0$)" % (title, str))
            ax[view].set_xlabel(r"$u_%d$" % (axis[0]))
            ax[view].set_ylabel(r"$u_%d$" % (axis[1]))
            ax[view].set_zlabel(r"$u_%d$" % (axis[2]))
```

```python
            ax[view].set_zlim(0, 1)
    global initialized
    initialized = True
    return


def _draw_path(s, **kwargs):
    for view in range(len(surface_index)):
        if xdim == 3:
            axis = (0,1,2)
        else:
            axis = [axis for axis in surface_axes if axis not in
                surface_index[view]]
        # for this view we draw the axes that are not set to zero
        x = G(s)
        if 'color' not in kwargs.keys():
            kwargs['color'] = 'k'
        if 'linewidth' not in kwargs.keys():
            kwargs['linewidth'] = 2
        ax[view].plot(x[:, axis[0]], x[:, axis[1]], x[:, axis[2]], **kwargs)
        ax[view].scatter3D(x[:, axis[0]][0], x[:, axis[1]][0], x[:,
            axis[2]][0], 'o', c='g', s=50, edgecolor='g') # starting point
        ax[view].scatter3D(x[:, axis[0]][-1], x[:, axis[1]][-1], x[:,
            axis[2]][-1], 'o', c='r', s=50, edgecolor='r') # ending point
    for parm_view in range(len(parm_index)):
        view = parm_view+len(surface_index)
        if sdim == 2:
            border = 0.01
            umax = np.max(s[:, 0])
            umax = umax + border if umax > 1.0 + border else 1.0 + border
            umin = np.min(s[:, 0])
            umin = umin - border if umin < 0.0 - border else 0.0 - border
            vmax = np.max(s[:, 1])
            vmax = vmax + border if vmax > 1.0 + border else 1.0 + border
            vmin = np.min(s[:, 1])
            vmin = vmin - border if vmin < 0.0 - border else 0.0 - border
            ax[view].set_xlim(umin, umax)
            ax[view].set_ylim(vmin, vmax)
            ax[view].plot(s[:, 0], s[:, 1], **kwargs)
            ax[view].plot(s[0, 0], s[0, 1], c='g', marker='o') # starting
                point
            ax[view].plot(s[-1, 0], s[-1, 1], c='r', marker='o') # ending
                point
            ax[view].grid(True)
        else:
```

```python
            axis = [axis for axis in parm_axes if axis not in
                parm_index[parm_view]]
            ax[view].plot(s[:, axis[0]], s[:, axis[1]], s[:, axis[2]],
                **kwargs)
            ax[view].scatter3D(s[0][axis[0]], s[0][axis[1]], s[0][axis[2]],
                'o', c='g', s=50, edgecolor='g') # starting point
            ax[view].scatter3D(s[-1][axis[0]], s[-1][axis[1]],
                s[-1][axis[2]], 'o', c='r', s=50, edgecolor='r') # ending
                point
    return

def draw_hold(s, title='CVA Solution', **kwargs):
    """
    cva.view.draw_hold(s, title='CVA Solution', **kwargs)

    The primary entry point cva.view.draw() is split into two parts,
    cva.view.draw_hold() and cva.view.draw_show(). This split makes
    it possible to construct graphics with multiple paths by placing
    one or more calls to cva.view.draw_hold() followed by a single call
    to cva.view.draw_show().

    Parameters
    ----------
    s : array_like
        This parameter contains a parameter space sequence that defines
        a path to be drawn.
    title : string, optional
        Specify the graphic's top title
    **kwargs : dictionary, optional
        This passthrough parameter allows some matplotlib options to be
        overridden. See the matplotlib documentation for usage information.

    Returns
    -------
    nothing

    Notes
    -----

    Examples
    --------
    import cva
    sa = (0.4,0.2)
```

```python
        sb = (0.5,0.5)
        cva.solve.select(cva.model.sphere,cva.metric.distance)
        path = cva.solve.run(sa,sb)
        cva.view.draw_hold(path)
        sa = (0.6,0.2)
        sb = (0.5,0.5)
        path = cva.solve.run(sa,sb)
        cva.view.draw_hold(path)
        cva.view.draw_show()

    In this example, two minimal paths are plotted on a single graphic.
    """
    if not initialized:
        _init(title)
        _draw_manifold()
    _draw_path(s, **kwargs)

def draw_show(title='', image_file=False, **kwargs):
    """
    cva.view.draw_show(image_file=False, **kwargs)

    The primary entry point cva.view.draw() is split into two parts,
    cva.view.draw_hold() and cva.view.draw_show(). This split makes
    it possible to construct graphics with multiple paths by placing
    one or more calls to cva.view.draw_hold() followed by a single call
    to cva.view.draw_show().

    Parameters
    ----------
    image_file : string (optional)
        This parameter contains the file name where an image is to be
        saved. The extension field of this name is checked to determine the
        format of the saved image.
    **kwargs : dictionary, optional
        This passthrough parameter allows some matplotlib options to be
        overridden. See the matplotlib documentation for usage information.

    Returns
    -------
    nothing

    Notes
    -----
    See cva.view.draw_hold for example usage.
```

```python
        """
        initialized = False
        plt.grid(True)
        if image_file:
            fig.tight_layout()
            fig.suptitle(title, y=1.0) # fix an inconsistency in matplotlib
            plt.savefig(image_file)
        else:
            plt.show()
        return


def draw(s, title='CVA Solution', image_file=False, **kwargs):
    """
    cva.view.draw(s, title='CVA Solution', image_file=False, **kwargs)

    This function is a matplotlib wrapper that can be used to display
    and save graphical representations of cva solutions.

    Parameters
    ----------
    s : array_like
        This parameter contains a parameter space sequence that defines
        a path to be drawn.
    title : string, optional
        Specify the graphic's top title
    image_file : string, optional
        In the default, a graphic is displayed. An image can also be
        directed to a file by giving its name.
        Example: image_file="mygraphic.pdf"
    **kwargs : dictionary, optional
        This passthrough parameter allows some matplotlib options to be
        overridden. See the matplotlib documentation for usage information.

    Returns
    -------
    nothing

    Notes
    -----
    This is the primary entry point for the cva.view module.

    Examples
    --------
    import cva
```

```
    sa = (0.4,0.2)
    sb = (0.6,0.6)
    cva.solve.select(cva.model.sphere,cva.metric.distance)
    path = cva.solve.run(sa,sb)
    cva.view.draw(path)

    In this example, the parameterization points sa and sb are mapped into
    their corresponding points on the surface of a unit sphere, and the
    resulting minimal curve is displayed.
    """
    cva.view.initialized = False
    draw_hold(s, title, **kwargs)
    draw_show(title, image_file)
    return

if __name__ == "__main__":

    print "running cva/view.py"
```

VITA

ROBERT WHITINGER

Education:	Bachelor of Science
	in Electrical Engineering
	University of Wisconsin
	Madison, Wisconsin 1971
	Master of Science
	in Mathematical Sciences
	East Tennessee State University
	Johnson City, Tennessee 2016
Honors Society:	Kappa Mu Epsilon
Professional Experience:	Director of Engineering, Amtelco
	Madison, Wisconsin 1973-1978
	Engineering Lead, Memory Systems Development
	Texas Instruments, Semiconductors
	Dallas, Texas 1978-1982
	Engineering Lead, Network Systems Development
	Texas Instruments, Industrial Systems
	Johnson City, Tennessee 1982-1990
	Lead Systems Architect , Siemens Inc.
	Johnson City, Tennessee 1990-1995
	International Product Manager, Siemens AG
	Nürnberg, Germany 1995-1997
	Senior Consulting Engineer, Siemens Inc.
	Johnson City, Tennessee 1997-2013

Licensures: Professional Engineer

                                   licensed to practice in the State of Tennessee

                 FAA Certified Flight Instructor

                                   Single/Multi-Engine Instrument

Publications: CT Chen, EM Benglas, SK Singh, DC Bullock,
and R Whitinger.
A novel probe tester for the characterization of
1 mbit/cm2 bubble memory devices.
Journal of Applied Physics, 52(3):23922394, 1981. [3]

R.J. Whitinger. Arbitrary drive for magnetic field
waveform control, August 23 1983.
US Patent 4,400,809. [19]