Title:

A Python Implementation of the Wilson-Fowler Spline for Open and Closed Curves

Author(s):

Rod W. Douglass and Laura M. Lang

Intended for:

3rd International Conference on Computational Methods in Engineering and Science

Los Alamos NATIONAL LABORATORY
EST.1943

A Python Implementation of the Wilson-Fowler Spline for Open and Closed Curves

Rod W. Douglass
XCP-1, MS T085
Los Alamos National Laboratory
Los Alamos, NM 87545
`rwd@lanl.gov`

Laura M. Lang
HPC-1, MS T085
Los Alamos National Laboratory
Los Alamos, NM 87545
`llang@lanl.gov`

Abstract[1]

We present a Python-class implementation of the Wilson-Fowler spline algorithm as outlined in [1] and as modified by Melvin [4]. That is, consider a set of N discrete points in two-dimensional space, $\{P_i = (x_i, y_i), i = 1, \ldots, N\}$ so that if $P_1 = P_N$, the resulting curve is closed. Define a set of continuous cubic Hermitian curves, $\Omega_s(u, v)$, for local (segment) coordinates (u, v) defined on each of the $s = 1, \ldots, N-1$ segments spanning P_s to P_{s+1}. The curve, Ω, is a Wilson-Fowler spline if [4]: for each $s = 1, \ldots, N-1$, Ω, restricted to the segment from P_s to P_{s+1}, is a member of Ω_s, Ω has a continuous slope and curvature, and Ω has tangent vectors, $\boldsymbol{T}_1$ at P_1 and $\boldsymbol{T}_N$ at P_N, if P forms an open curve. For closed curves, the tangents are found by applying the algorithm to the first/last point, forming a "cyclic" non-linear algebraic system. The non-linear algebra problem for the tangents at each node is solved using Newton's method with a line-search update (based on pg. 383, ff. [11]).

Once the spline is computed, methods were written to perform operations on the spline. These include ray-spline intersection, insertion of m-points between original points, and point re-distribution methods. Point insertion and re-distribution may use either equal arc-length, equal angles, or equal Euclidean length between points as criteria for point placement.

Results demonstrate creation of the spline and ray-spline intersections for two test problems.

1 Introduction

The Wilson-Fowler spline (subsequently referred to here as WFS) [1] was introduced in the mid 1960's as a means of passing a smooth curve through an ordered set of N-points in a plane. The goal was to produce a mathematical approximation to an infinitely thin elastic spline passing through the points, settling upon a set of cubic polynomials defined over each of the $N-1$ intervals between each pair of points such that the curvature at the end points of the intervals matched those from their

[1] The title and text have been approved for unrestricted release as Los Alamos National Laboratory report LA-UR 11-00000.

neighboring interval. Much has been written concerning this spline including properties of the spline by Emery [3], an error analysis of the spline and some of its other properties with a reformulation of the solution algorithm by Melvin [4], a comprehensive annotated history of the WFS by Fritsch [7] through 1985, comparison of the WFS to other interpolating splines [6], and a method of converting the WFS to a cubic B-spline [8]. Since 1984 [5], the WFS has been adopted by the US Department of Energy Nuclear Weapons Complex as the primary means of transfer of data between CAD/CAM systems. That is, drawings should use only the WFS and should include a table of node points and end node slopes. For digital applications, IGES was the recommended framework. Conversion of the WFS to a global IGES framework was reported by Dolan [9].

We present a specific instantiation of the WFS algorithm as described by Melvin [4], but with some of the features used in the original paper by Fowler and Wilson. That is, we use Melvin's end node slope specification but allow the user to use the slope estimator described in the original paper and appendix [1] for cases where there is no specific end-slope requirements. We also allow the points to form a closed curve in a plane, something neither paper considers. In particular, the algorithm is written as a Python class and uses some Numpy and Scipy methods. The class holds the basic geometric information required to compute the nodal tangents as well as those tangents. In addition the class computes the Euclidean length of each segment (*i.e.*, the interval between consecutive points), the arc-length of each segment, and the total Euclidean and arc-lengths. A Ray class is also created which defines a ray as origin and direction tuples. A ray can then be cast against the spline using ray-box intersection theory to locate the segment being intersected, that segment being further examined for roots of a cubic equation describing the ray-spline intersection, and finally returning the x,y-coordinate tuple as well as distance from the ray origin to the intersection point, and the unit normal and tangent tuples at each intersection. The ray-spline intersection capability is used to find coordinates for refinement (addition) of nodes between original node pairs or for a new distribution of M-points along the spline using either equal angle or arc-length as the redistribution basis.

In the following sections, a synopsis of the WFS will be given and the algorithm to compute it described for both open and closed curves. The WFS Python class will be described and then used on selected test problems to illustrate its utility. Concluding remarks will summarize the results.

2 The Wilson-Fowler Spline

Consider the N ordered points, $\{P_i = (x_i, y_i), i = 1, \ldots, N\}$, shown in Figure 1 which lie in a Cartesian plane. If $P_1 = P_N$, the points describe a closed curve; otherwise they describe an open curve. Let $E = N-1$ segments be defined as the interval between consecutive pairs of points, $\{S_j = (P_j, P_{j+1}), j = 1, \ldots, E\}$. Then, for segment j, define a local coordinate system $(u, v), 0 \leq u, v \leq 1$ having unit vectors $(\hat{U}, \hat{V})$, with translated origin at P_j, $\hat{U}$ parallel to the line connecting nodes j and $j+1$, and which is a rotation of the $\mathbf{x} = (x, y)$ coordinates through an angle $\theta_j = \tan^{-1}(\frac{y_{j+1}-y_j}{x_{j+1}-x_j})$ so that

$$\begin{Bmatrix} x \\ y \end{Bmatrix} = \begin{Bmatrix} x_j \\ y_j \end{Bmatrix} + L_j \begin{bmatrix} \cos(\theta_j) & -\sin(\theta_j) \\ \sin(\theta_j) & \cos(\theta_j) \end{bmatrix} \begin{Bmatrix} u \\ v \end{Bmatrix}. \quad (1)$$

For that segment, define a curve Ω_j such that

$$\Omega_j = \{P = P_j + L_j[u\hat{U}_j + v(u)\hat{V}_j], \text{where } v(u) = \tau_{a_j} u(u-1) + \tau_{b_j} u^2(u-1)\} \quad (2)$$

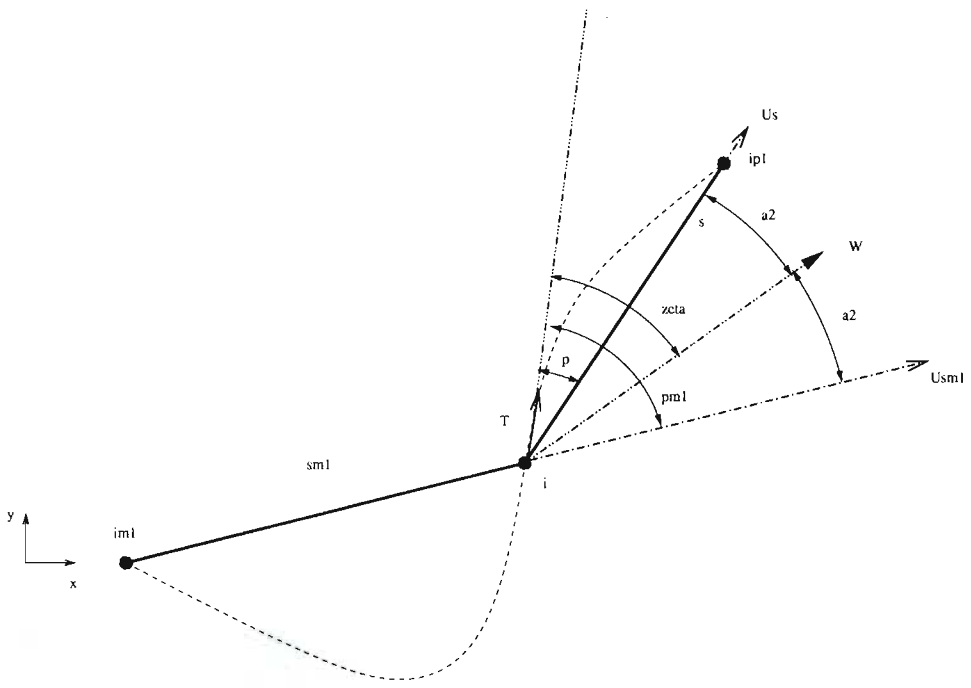

Fig. 1: For segment s, the unit tangent vector, $\hat{T}_i$, at node i is represented relative to the unit vector, $\hat{W}_i = W_i/|W_i|, W_i = (\hat{U}_{i-1} + \hat{U}_i)/2$ as having an angle $\phi_{i-1} = \zeta_i + \alpha_i/2$ or, equivalently, $\phi_i = \zeta_i - \alpha_i/2$ so that $\tau_{a_i} = \tan(\phi_i) = \tan(\zeta_i - \alpha_i/2)$ and $\tau_{b_{i-1}} = \tan(\phi_{i-1}) = \tan(\zeta_i + \alpha_i/2)$.

In order to understand how τ_a and τ_b are related to the slope of the spline, consider the derivatives within any given segment

$$\frac{dx}{du} = L\left[\cos(\theta) - \sin(\theta)\,\frac{dv(u)}{du}\right] \tag{3}$$

$$\frac{dy}{du} = L\left[\sin(\theta) + \cos(\theta)\,\frac{dv(u)}{du}\right] \tag{4}$$

where

$$\frac{dv(u)}{du} = (3u-1)(u-1)\,\tau_a + u(3u-2)\tau_b \tag{5}$$

and

$$\frac{d^2v(u)}{du^2} = 2\left((3u-2)\tau_a + (3u-1)\tau_b\right). \tag{6}$$

Then, at the beginning of the segment (*i.e.*, at $u = 0$), we have

$$\left.\frac{dy}{dx}\right|_{u=0} = \left.\frac{dy/du}{dx/du}\right|_{u=0} = \frac{L(\sin(\theta) + \tau_a\cos(\theta))}{L(\cos(\theta) - \tau_a\sin(\theta))} = \frac{\tan(\theta) + \tau_a}{1 - \tau_a\tan(\theta)}, \tag{7}$$

so that if $\tau_a = \tan(\phi_a)$, we have

$$\left.\frac{dy}{dx}\right|_{u=0} = \frac{\tan(\theta) + \tan(\phi_a)}{1 - \tan(\phi_a)\tan(\theta)} = \tan(\theta + \phi_a). \tag{8}$$

Then, $\phi_a = \tan^{-1}\left(\left.\frac{dy}{dx}\right|_{u=0}\right) - \theta$. Likewise, at the end of the segment (*i.e.*, at $u = 1$), $\tau_b = \tan(\phi_b)$ and $\phi_b = \tan^{-1}\left(\left.\frac{dy}{dx}\right|_{u=1}\right) - \theta$.

The curvature of the spline, $K(u)$, is defined as

$$\begin{aligned} K(u) = \frac{\left|\epsilon_{ijk}\frac{dx_j}{du}\frac{d^2x_k}{du^2}\right|}{\left|\frac{dx_i}{du}\right|^3} &= \frac{\frac{dx}{du}\frac{d^2y}{du^2} - \frac{dy}{du}\frac{d^2x}{du^2}}{\left[\left(\frac{dx}{du}\right)^2 + \left(\frac{dy}{du}\right)^2\right]^{3/2}} \\ &= \frac{1}{L}\frac{\frac{d^2v}{du^2}}{\left[1 + \left(\frac{dv}{du}\right)^2\right]^{3/2}} \\ &= \frac{2}{L}\frac{(3u-2)\tau_a + (3u-1)\tau_b}{[1 + ((3u-1)(u-1)\tau_a + u(3u-2)\tau_b)^2]^{3/2}} \end{aligned} \tag{9}$$

Then for node j, which is the end of segment $j-1$ and also the beginning of segment j, we have that the curvatures from the two segments are set equal to insure continuity of curvature from segment to segment. Then

$$K_{j-1}(1) = \frac{2}{L_{j-1}}\frac{\tau_{a_{j-1}} + 2\tau_{b_{j-1}}}{[1 + \tau_{b_{j-1}}^2]^{3/2}} \tag{10}$$

$$K_j(0) = \frac{-2}{L_j}\frac{2\tau_{a_j} + \tau_{b_j}}{[1 + \tau_{a_j}^2]^{3/2}} \tag{11}$$

or,

$$\begin{aligned} \mathcal{F}_j = L_{j-1}(1 + D_jZ_j)^2 \; &[2(Z_j - D_j) + \tau_{b_j}(1 + D_jZ_j)] \\ &+ L_j(1 - D_jZ_j)^2 \left[2(Z_j + D_j) + \tau_{a_{j-1}}(1 - D_jZ_j)\right] = 0, \end{aligned} \tag{12}$$

where, referring to Figure 1,

$$D_j = \tan(\alpha_j/2) \tag{13}$$

$$Z_j = \tan(\zeta_j) \tag{14}$$

$$\tau_{a_j} = \frac{Z_j - D_j}{1 + Z_jD_j} \tag{15}$$

and

$$\tau_{b_{j-1}} = \frac{Z_j + D_j}{1 - Z_jD_j}. \tag{16}$$

Then with

$$\tau_{a_{j-1}} = \frac{Z_{j-1} - D_{j-1}}{1 + Z_{j-1}D_{j-1}} \tag{17}$$

and

$$\tau_{b_j} = \frac{Z_{j+1} + D_{j+1}}{1 - Z_{j+1}D_{j+1}}, \tag{18}$$

we have that Equation 12 is a non-linear algebraic equation for Z_j which depends only on Z_{j-1} and Z_{j+1}. Then, writing a similar equation for all nodes, $2 \leq j \leq N-1$ gives a set of $N-2$ non-linear equations to be solved for Z_j, provided boundary values for τ_{a_1} and $\tau_{b_{N-1}}$ are provided.

2.1 Solving for Z_j

The j-th equation for Z, Equation 12, can be written as

$$\mathcal{F}_j(Z_{j-1}, Z_j, Z_{j+1}) = 0 \approx \mathcal{F}_j(\tilde{Z}_{j-1}, \tilde{Z}_j, \tilde{Z}_{j+1}) + \widetilde{\frac{\partial \mathcal{F}_j}{\partial Z_i}}(Z_i - \tilde{Z}_i), \tag{19}$$

so that Newton's method may be used to iteratively solve for Z. That is,

$$\tilde{\mathcal{J}}_{ij} \Delta Z_j \approx -\mathcal{F}_j(\tilde{Z}_{j-1}, \tilde{Z}_j, \tilde{Z}_{j+1}), \tag{20}$$

is iteratively solved for ΔZ_j by inverting the Jacobian,

$$\begin{aligned} \mathcal{J}_{ij} &= \frac{\partial \mathcal{F}_i}{\partial Z_j}, \\ &= \begin{cases} \mathcal{J}_{i,j-1}, \mathcal{J}_{i,j}, \mathcal{J}_{i,j+1}, \text{for } j = i \\ \quad 0, \text{otherwise} \end{cases} \end{aligned} \tag{21}$$

where

$$\mathcal{J}_{i,i-1} = L_i \frac{(1 - Z_i D_i)^3 (1 + D_{i-1}^2)}{(1 + Z_{i-1} D_{i-1})^2} \tag{22}$$

$$\begin{aligned} \mathcal{J}_{i,i} = L_i &\left[(1 - Z_i D_i)^2 (2 - 3 D_i \tau_{a_{i-1}}) - 4 D_i (Z_i + D_i)(1 - Z_i D_i)\right] \\ &+ L_{i-1} \left[(1 + Z_i D_i)^2 (2 + 3 D_i \tau_{b_i}) + 4 D_i (Z_i - D_i)(1 + Z_i D_i)\right] \end{aligned} \tag{23}$$

$$\mathcal{J}_{i,i+1} = L_{i-1} \frac{(1 + Z_i D_i)^3 (1 + D_{i+1}^2)}{(1 - Z_{i+1} D_{i+1})^2}. \tag{24}$$

The Jacobian is a tri-diagonal matrix and $\tilde{()}$-variables are prior iteration values or initial guesses. Matrix inversion is done using the Tri-Diagonal Matrix Algorithm (or the Thomas algorithm) (for example, [11], pp. 50-51) which does not destroy the input matrix data. In some instances, a simple update of the form $Z_j^{n+1} = Z_j^n + \Delta Z_j,\ n = 0, \ldots$ may become unstable, even if a weight-factor is applied to ΔZ_j. Consequently, a Newton iteration with line searching and backtracking was implemented, based upon the scheme described in [11], pp. 384-389.

2.2 Boundary conditions

If $P_1 \neq P_N$, slopes are required at those points in order to solve the equations for Z. Wilson and Fowler [1] specify that if a specific slope is not available, that $\tau_{a_1} = -\tau_{b_1}$ and/or $\tau_{b_{N-1}} = -\tau_{a_{N-1}}$. In [1], pg. 90, Wilson and Fowler modify these conditions to allow for the curvature at the two adjacent nodes and the segment length of the adjacent two segments,

$$\tau_{a_1} = -\tau_{b_1} + \frac{C_{1,2}(\tau_{a_2} + \tau_{b_2})}{[1 + |\tau_{a_2}|\,|\tau_{b_2}|]^2} \tag{25}$$

$$\tau_{b_{N-1}} = -\tau_{a_{N-1}} + \frac{C_{N-1,N-2}(\tau_{a_{N-2}} + \tau_{b_{N-2}})}{\left[1 + \left|\tau_{a_{N-2}}\right|\left|\tau_{b_{N-2}}\right|\right]^2}, \tag{26}$$

where

$$C_{i,j} = \max\left(\left(\frac{L_i}{L_j}\right)(1.5 - L_i/L_j) - 0.02, 0\right). \tag{27}$$

If, instead, slopes are available at the end nodes, then they are related to the τ's as follows. Let $\hat{T}_1 = (\hat{T}_{x_1}, \hat{T}_{y_1})$ be a unit vector parallel to the desired slope (*i.e.*, $dy/dx|_{P_1}$). Then

$$\left.\frac{dy}{dx}\right|_{P_1} = \frac{\tan(\theta) + \tau_{a_1}}{1 - \tan(\theta)\tau_{a_1}} = \frac{\hat{T}_{y_1}}{\hat{T}_{x_1}}, \tag{28}$$

with $\tan(\theta) = (y_{P_2} - y_{P_1})/(x_{P_2} - x_{P_1}) = \Delta y_1/\Delta x_1$, and solving for τ_{a_1} gives

$$\tau_{a_1} = \frac{\Delta x_1 \hat{T}_{y_1} - \Delta y_1 \hat{T}_{x_1}}{\Delta x_1 \hat{T}_{x_1} + \Delta y_1 \hat{T}_{y_1}}. \tag{29}$$

Likewise, if $\hat{T}_N = (\hat{T}_{x_N}, \hat{T}_{y_N})$ is a unit vector parallel to the slope at P_N, then

$$\tau_{b_{N-1}} = \frac{\Delta x_{N-1} \hat{T}_{y_N} - \Delta y_{N-1} \hat{T}_{x_N}}{\Delta x_{N-1} \hat{T}_{x_N} + \Delta y_{N-1} \hat{T}_{y_N}}. \tag{30}$$

2.3 Closed Curve Formulation

If $P_1 = P_N$, the curve is closed and Equation 12 applies to each of the $1, \ldots, N$ nodes. However, $Z_1 = Z_N$, so that there are $N - 1$ independent equations to be solved. The equation for node 1 becomes

$$\begin{aligned} &L_{N-1}(1 + D_1 Z_1)^2 \left[2(Z_1 - D_1) + \tau_{b_1}(1 + D_1 Z_1)\right] \\ &+L_1(1 - D_1 Z_1)^2 \left[2(Z_1 + D_1) + \tau_{a_{N-1}}(1 - D_1 Z_1)\right] = 0, \end{aligned} \tag{31}$$

where $\tau_{a_{N-1}} = (Z_{N-1} - D_{N-1})/(1 + Z_{N-1}D_{N-1})$. For node $N - 1$, the equation becomes

$$\begin{aligned} &L_{N-2}(1 + D_{N-1}Z_{N-2})^2 \left[2(Z_{N-2} - D_{N-2}) + \tau_{b_{N-1}}(1 + D_{N-1}Z_{N-1})\right] \\ &+L_{N-1}(1 - D_{N-1}Z_{N-1})^2 \left[2(Z_{N-1} + D_{N-1}) + \tau_{a_{N-2}}(1 - D_{N-1}Z_{N-1})\right] = 0, \end{aligned} \tag{32}$$

but $\tau_{b_{N-1}} = (Z_1 + D_1)/(1 - Z_1 D_1)$.

The Jacobian matrix is now nearly tri-diagonal, but has additional entries in the last column of row 1 (*i.e.*, $\mathcal{J}_{1,0}$) and the first column of row $N - 1$ (*i.e.*, $\mathcal{J}_{N-1,N}$), forming a cyclic tri-diagonal matrix problem. Solution to such systems is described in [11] (pp. 74-75) and implemented here.

2.4 Ray-WFS Intersection

A ray can be used to sample a WFS to extract information as to the number of intersections, their coordinates, unit normal and tangent vectors, slopes, and curvatures at the intersection point or points. There are two levels of review to determine if there is a ray-WFS intersection. First, the spline is scanned segment by segment to determine if its bounding box is intersected by the ray using the ray-box intersection algorithm as described by Glassner [10] (pp. 65-67). Those segments whose bounding boxes are intersected are then candidates for further examination to see if at least one intersection truly occurs. For those candidate cases, a typical situation is shown in Figure 2.

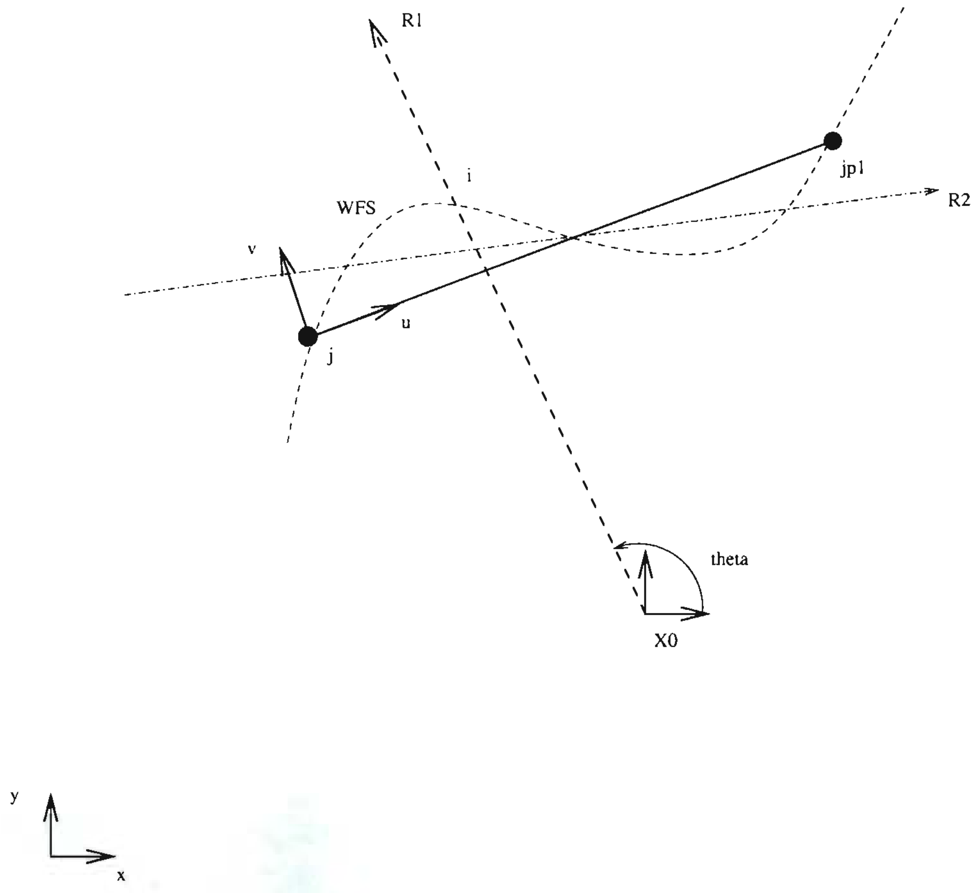

Fig. 2: Ray_1, centered at (X_o, Y_o) and at an angle $\theta = \tan^{-1}(D_y/D_x)$, intersects the segment-j WFS at $u = u^*$ once. Ray_2 is an example of a ray intersecting a WFS three times in one segment.

A ray object (*c.f.*, Section 3.2), is defined by an origin tuple, $\mathbf{X}_o = (X_o, Y_o)$, and a direction unit vector tuple, $\mathbf{D} = (D_x, D_y)$, and is expressed parametrically as

$$\mathbf{x} = (x, y) = \mathbf{X}_o + g\mathbf{D}, \text{ for } g \geq 0. \tag{33}$$

Simply stated, an intersection exists if the value of $u = u^*$ obtained by equating this equation for the ray (Equation 33) to the equation for the WFS in that segment, Equation 1, lies in the range

$0 \leq u^* \leq 1$, that is, u^* is a valid root. Equating these equations and solving for u results in the cubic polynomial (with real coefficients):

$$pu^3 + qu^2 + ru + s = 0, \tag{34}$$

where

$$p = (\tau_{a_j} + \tau_{b_j})A \tag{35}$$

$$q = -(2\tau_{a_j} + \tau_{b_j})A \tag{36}$$

$$r = B + A\tau_{a_j} \tag{37}$$

$$s = y_j - Y_o - \frac{\Delta y_o}{\Delta x_o}(x_j - X_o) \tag{38}$$

and

$$A = x_{j+1} - x_j + \frac{\Delta y_o}{\Delta x_o}(y_{j+1} - y_j) \tag{39}$$

$$B = y_{j+1} - y_j - \frac{\Delta y_o}{\Delta x_o}(x_{j+1} - x_j). \tag{40}$$

The exact solution for a cubic polynomial having real coefficients (*e.g.*, [2], pp. 104-105) is used to find the value of $u = u^*$ for the intersection. Potentially, there are up to 3 real valid roots. Then, the coordinates of the intersection are determined by substitution of each u^* into Equations 1, the slope using Equations 3 and 4, the curvature using Equation 9, and the unit tangent and principle unit normal vectors from

$$\hat{\mathbf{t}} = \frac{\left((\cos(\theta) - \sin(\theta)\frac{dv}{du}), (\sin(\theta) + \cos(\theta)\frac{dv}{du})\right)}{\left[1 + (\frac{dv}{du})^2\right]^{1/2}} = (\hat{t}_x, \hat{t}_y) \tag{41}$$

$$\hat{\mathbf{n}} = \frac{\left(-(\sin(\theta) + \cos(\theta)\frac{dv}{du}), (\cos(\theta) - \sin(\theta)\frac{dv}{du})\right)}{\left[1 + (\frac{dv}{du})^2\right]^{1/2}} = (-\hat{t}_y, \hat{t}_x). \tag{42}$$

The results for each valid intersection are appended to the appropriate variables in the Ray class.

2.5 Arc Length and Strain Energy

The arc-length along the spline as measured from the beginning node of segment j, $u = 0$, to $u = u_o$ is

$$s_j(u_o) = \int_{u=0}^{u_o} \left| \frac{d\mathbf{x}_j}{du} \cdot \frac{d\mathbf{x}_j}{du} \right| du$$
$$= L_j \int_{u=0}^{u_o} \left[1 + \left(\frac{dv_j}{du}\right)^2\right]^{1/2} du, \tag{43}$$

which was evaluated using the scipy.integrate.quad method.

Likewise, the strain energy in the spline is a measure of its roughness. Since the WFS passes a smooth curve through the points given, the roughness is manifest through the curvature. That is, the strain energy is defined as

$$\text{strain energy} = \mathcal{E} = \int_{s=0}^{\mathcal{L}} K^2(s)\, ds = \sum_{j=1}^{E} \int_{u=0}^{1} K_j^2(u) \left| \frac{d\mathbf{x}_j}{du} \cdot \frac{d\mathbf{x}_j}{du} \right| du \tag{44}$$

where $\mathcal{L}$ is the total arc-length of the curve and K is the curvature defined in Equation 9. These integrals are also computed using the scipy.integrate.quad method.

3 The WilsonFowler Python Class

The Wilson-Fowler algorithm just described is implemented in Python and is instantiated using the WF class. The class contains 28 methods, 13 of which are used in completing the spline calculations, the balance being methods that either are operations on a spline once calculated or general utility methods. In addition, two additional Python classes are provided to support some of the operations on a spline, in particular those that intersect a ray with the spline. The first is a Ray-class containing a pair of tuples, the first being the ray origin coordinates and the second a tuple representing the unit vector pair which is parallel to the ray. In addition, the class holds any ray-intersection data generated when a ray intersects a spline, including the number of times the ray crosses the spline, the x,y-coordinates of the intersection point, the slope, curvature, and unit normal and tangents to the spline at the intersection point or points. The second class is a simple Box-class containing a tuple of tuples of the minimum and maximum coordinate pairs for a given segment. Boxes are used to efficiently [?] locate the segment of the spline through which the ray crosses.

3.1 WF Class Methods

To instantiate a WF object, the user creates a list of x-coordinates and associated y-coordinates and determines if a specific beginning or ending slope must be imposed on the spline. The default behavior is that the end-slopes will not be specified but that they will be determined by the algorithm according to the methods described in Section 2.2. If the curve is a closed curve, this will be detected by the algorithm and end-slopes are, of course, unnecessary. If one or both end-slopes of an open curve are to be specified, create a unit vector parallel to the desired slope and pass those values into the object. A example Python session might look like:

Example: Instantiation of WF object.

```
>>> from WilsonFowler import *
>>> # Assume $y = x * *2$
>>> x = [0.,1.,2.,3.]
>>> y = [X*X for X in x]
>>> # Use defaults
>>> spline = WF(x, y)
>>> # Print $\tau_a$ and $\tau_b$ for the spline
>>> spline.ta
array([-0.22469215, -0.27270126, 0.00850517])
>>> spline.tb
array([ 0.20002518, 0.13364725, -0.00850517])
>>> # Check the slope at the end of the spline
>>> # segmment 2, $u = 1$.
>>> data = spline.getdata(2, 1.)
>>> slope = data[1]
>>> slope
4.7878859570445966
>>> # Use specified slope at the end-node of the spline
>>> # $dy/dx|_3 = 2x|_3 = 6$
>>> $T_x = 1.$
>>> $T_y = 6.$
>>> # or
>>> from math import sqrt
>>> $t_x$ = 1./sqrt(37.)
>>> $t_y$ = 6./sqrt(37.)
>>> spline2 = WF(x, y, setEndSlope = True, endTx = $t_x$, endTy = $t_y$)
>>> spline2.ta
array([ -0.22889108, -0.26967566, -0.00027224])
>>> spline2.tb
array([ 0.203055, 0.12472351, 0.03225806])
>>> data2 = spline2.getdata(2, 1.)
>>> slope2 = data2[1]
>>> slope2
6.0000000000000009
```

Methods supporting instantiation of a WF object include:

__init__(self, x, y, useLineSearch = True, setBeginSlope = False, beginTx = 0.0, beginTy = 1.0, setEndSlope = False, endTx = 0.0, endTy = -1.0, oldSlopeBegin = False, oldSlopeEnd = False)

Description : This is the constructor for the WF-class. It contains all of the geometric data and post-Newton iteration spline data needed to describe the spline.

Methods Called : buildSpline, sgn, FofZ, calcTotalArclength, strainEnergy

Input Data:

x,y : (list, list) x and y coordinates of ordered points comprising the spline
useLineSearch : (Boolean) True implies that the line-search method for updating the solution during iteration is used, False implies that the simpler weighted update method be used. Default = True.
setBeginSlope : (Boolean) True implies specified begin slope of the spline. Default = False.
beginTx, beginTy : (scalar, scalar) x and y components of unit vector parallel to beginning slope. Default = 0., 1.
setEndSlope : (Boolean) True implies specified end slope of the spline. Default = False.
endTx, endTy : (scalar, scalar) x and y components of unit vector parallel to end slope. Default = 0., -1.
maxIter : (integer) maximum number of Newton iterations allowed. Default = 20
oldSlopeBegin : (Boolean) If True, use simpler first node boundary condition: $\tau_a = -\tau_b$. Default: False (*i.e.*, use Equation 25)
oldSlopeEnd : (Boolean) If True, use simpler last node boundary condition: $\tau_b = -\tau_a$. Default: False (*i.e.*, use Equation 26)

buildSpline (self, maxIter = 20, useLineSearch = True)

Description : This method does all the Newton iteration calculations to find the τ_a and τ_b for the spline which satisfies the end slope conditions.
Methods Called : FofZ, Jacobian, cyclicSolve, TDMA, lineSearch, L2norm
Input Data:
maxIter : (integer) maximum number of Newton iterations allowed. Default = 20
useLineSearch : (Boolean) True implies that the line-search method for updating the solution during iteration is used, False implies that the simpler weighted update method be used. Default = True.
Return : Boolean flag: True if a spline successfully constructed, False otherwise. Updates the lists: self.Z, self.ta, self.tb, and the flags self.status and self.check

FofZ (self, x)

Description : Given a list, x, of Z-values find $F(Z)$ (*c.f.*, Equation 12).
Methods Called : dotprod
Input Data:
x : (list) of Z-values as in Equation 12
Return : $(\mathrm{F} \cdot \mathrm{F})/2$. Updates the list, self.F

strainEnergy (self)

Description : For each segment calculate its strain energy using Equation 44, storing it in the list self.Energy, and accumulate the sum, storing it in self.totalEnergy.
Methods Called : SciPy.integrate.quad
Input Data:
none :
Return : Nothing, Updates the list self.Energy and scalar self.totalEnergy

calcTotalArcLength (self)

Description : For each segment calculate its arc length, storing it in the list self.segArcLength, and accumulate the sum, storing it in self.totalArclength.
Methods Called : segmentArcLength
Input Data:
none :
Return : Nothing, Updates the list, self.segArcLength and scalar self.totalArcLength

sgn (self, A)

Description : Given a scalar, return its sign: $+1$ if $|A| \geq 0.$ or -1 otherwise.
Methods Called : None
Input Data:
A : (scalar)
Return : ± 1

Jacobian (self, x)

Description : Computes the elements of the Jacobian matrix, Eq. 21.
Methods Called : None
Input Data:
x : (list) current Z-values
Return : Lower diagonal list (Eq. 22), the main diagonal list (Eq. 23), and upper diagonal list (Eq. 23), $\alpha = \mathcal{J}_{N-1,N}$ and $\beta = \mathcal{J}_{1,0}$.

TDMA (self, A, B, C, D)

Description : Solve a tri-diagonal linear algebra system.
Methods Called : None
Input Data:
A : (list) lower diagonal entries in the linear algebra problem
B : (list) main diagonal entries in the linear algebra problem
C : (list) upper diagonal entries in the linear algebra problem
D : (list) right-hand-side entries in the linear algebra problem
Return : the solution list

cyclicSolve (self, A, B, C, Alpha, Beta, D)

Description : Solve a cyclic tri-diagonal linear algebra system. Used only for closed curves.
Methods Called : TDMA, dotprod
Input Data:
A : (list) lower diagonal entries in the linear algebra problem
B : (list) main diagonal entries in the linear algebra problem
C : (list) upper diagonal entries in the linear algebra problem
Alpha : (scalar) $\mathcal{J}_{N-1,N}$
Beta : (scalar) $\mathcal{J}_{1,0}$
D : (list) right-hand-side entries in the linear algebra problem
Return : the solution list

lineSearch (self, xold, fold, g, stpmax, p)

Description : Updates the Newton iterates for Z using a line-search algorithm with backtrack, if needed (*c.f.*, [11], pp. 383-389).
Methods Called : dotprod, FofZ
Input Data:
  xold : (list) the values from the prior Newton iteration needing to be updated.
  fold : (scalar) the values of $F_iF_i/2$ for the prior Newton iteration
  g : (list) the gradient of self.F: $g_j = F_i\mathcal{J}_{i,j}$
  stpmax : (scalar) limit to the length of step available for update to avoid evaluating F at undefined points.
  p : (list) the current Newton step values of ΔZ from either cyclicSolve or TDMA
Return : the updated solution list, associated value of $F_iF_i/2$

L2norm (self, x)

Description : Find the L_2-norm of the list, x = $\sqrt{x_ix_i}$.
Methods Called : dotprod
Input Data:
  x : (list)
Return : (scalar) $\sqrt{x_ix_i}$

dotprod (self, x)

Description : Find the dot product of the list, x = x_ix_i.
Methods Called : None
Input Data:
  x : (list)
Return : (scalar) x_ix_i

SUM (self, x)

Description : Find the sum of entries in the list, x = $\sum_{i=1}^{N} x_i$.
Methods Called : None
Input Data:
  x : (list)
Return : (scalar) $\sum_{i=1}^{N} x_i$

Utility methods include:

__str__(self)

Description : Return a string representation of the WF-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinate pairs for the spline, and τ_a and τ_b.

__repr__(self)

Description : Return a string representation of the WF-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinate pairs for the spline, and τ_a and τ_b.

writeXY (self, fileName, xy = None)

Description : Write the coordinates of the spline to a file named fileName if called with only the file name. Otherwise, write the given xy tuple to the file named fileName.
Methods Called : None
Input Data:
  fileName : (string) the name of the file to open, write to, and close
  xy : (scalar tuple) x and y-coordinate pairs
Return : Nothing.

cbrt (self, A)

Description : Return the cube root of a scalar
Methods Called : None
Input Data:
  fileName : (scalar)
Return : $A^{1/3}, A \geq 0$ or $-|A|^{1/3}, A < 0$

Finally, there are methods provided which operate on a given spline. In particular, a spline may be refined by adding points between existing points such that they fall on the spline, or a spline may be used to distribute newN points along the spline, where newN may be greater than, equal to, or less than N, or a spline may have a ray cast against it. Refinement and distribution may be done using one of three strategies: 1.) equal Euclidean length between points, 2.) equal angle between points, and 3.) equal arc-length between points. Ray-casting is used, for example, in equal angle distribution or refinement.

getRayIntersection (self, ray)

Description : Given a ray emanating from an origin, $R_0 = (R_{0_x}, R_{0_y})$ and in a specified direction $R_d = (R_{d_x}, R_{d_y})$, find its intersection with the calling spline. The equation for the ray is $y - R_{0_y} = R_{d_y}/R_{d_x}(x - R_{0_x})$. The spline is scanned segment by segment until the segment is found which has a ray-intersection. Then, within that segment, the value of u is found which satisfies the resulting equation when the ray equation (just above) is substituted into Eqs. 1. All segments are scanned to find all ray-spline crossings.
Methods Called : Segment.intersect, Ray.push, findCrossing, getdata
Input Data:
  ray : (Ray object) see Section 3.2
Return : (Boolean) True, if at least one crossing is found, False otherwise. Intersection results are stored in the ray-object.

Distrib (self, newN)

Description : Distribute newN points along the calling spline requiring equal Euclidean length between the new points, retaining the first and last nodes untouched.
Methods Called : getxy
Input Data:
  newN : (integer) the new number of points desired
Return : (tuple of x-list, y-list) the new coordinates corresponding to newN locations

equalAngleDistrib (self, newN, origin)

Description : Distribute newN points along the calling spline requiring equal angular measure relative to the origin specified between new points, retaining the first and last nodes untouched.
Methods Called : getRayIntersection, creates a Ray object
Input Data:
  newN : (integer) the new number of points desired
  origin : (tuple) x,y-coordinates to center the angular measure
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

equalAngleRefine (self, newN, origin)

Description : Add newN points between existing nodes along the calling spline requiring equal angular measure relative to the origin specified between the new points, retaining the original nodes untouched. This results in $\mathrm{N} + \mathrm{newN}(\mathrm{N} - 1)$ nodes.
Methods Called : getRayIntersection, creates a Ray object
Input Data:
  newN : (integer) the new number of points desired
  origin : (tuple) x,y-coordinates to center the angular measure
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

equalArcLengthDistrib (self, newN)

Description : Distribute newN points along the calling spline requiring equal arc length between new points, retaining the first and last nodes untouched.
Methods Called : uFromArcLength, getxy
Input Data:
  newN : (integer) the new number of points desired
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

equalArcLengthRefine (self, newN)

Description : Add newN points between existing nodes along the calling spline requiring equal arc length between the new points, retaining the original nodes untouched. This results in $\mathrm{N} + \mathrm{newN}(\mathrm{N} - 1)$ nodes.
Methods Called : uFromArcLength, getxy
Input Data:
  newN : (integer) the new number of points desired
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

refine (self, newN)

Description : Add newN points between existing nodes along the calling spline requiring equal increments in u between the new points, retaining the original nodes untouched. This results in $\mathrm{N} + \mathrm{newN}(\mathrm{N} - 1)$ nodes.
Methods Called : getxy
Input Data:
newN : (integer) the new number of points desired
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

Refine (self, newN)

Description : Add newN points between existing nodes along the calling spline requiring equal increments in Euclidean length between the new points, retaining the original nodes untouched. This results in $\mathrm{N} + \mathrm{newN}(\mathrm{N} - 1)$ nodes. In principle, equivalent to the refine method.
Methods Called : getxy
Input Data:
newN : (integer) the new number of points desired
Return : (tuple of x-list, y-list) the new coordinates corresponding to new locations

These modification methods are supported by additional methods:

findCrossing (self, segment, ray)

Description : In the segment, find u such that the ray intersects the calling spline.
Methods Called : cubicSolve
Input Data:
segment : (integer) the segment for which the ray crosses
ray : (Ray object) the ray intersecting the spline
Return : (Boolean) True if a root is found, False otherwise, (integer) number of roots to the cubic found (usually 1), and (scalar) u

getdata (self, segment, u)

Description : In the segment, compute the values of x, y-coordinate tuple, slope, curvature, and unit normal and unit tangent tuples corresponding to the given u.
Methods Called : None
Input Data:
segment : (integer) the desired segment
u : (scalar) the value of u for which the spline data are desired
Return : (tuple) coordinates, (scalar) slope, (scalar) curvature, (tuple) unit normal, (tuple) unit tangent

getxy (self, segment, u)

Description : In the segment, compute the values of x, y-coordinate tuple corresponding to the given u.
Methods Called : None
Input Data:
- **segment :** (integer) the desired segment
- **u :** (scalar) the value of u for which the spline data are desired

Return : (tuple) coordinates

uFromArcLength (self, segment, arcL)

Description : In the segment find u corresponding to the given arcL. This is the inverse to the method segmentArcLength.
Methods Called : segmentArcLength
Input Data:
- **segment :** (integer) the desired segment
- **arcL :** (scalar) the value of arc length within segment for which u is desired

Return : (scalar) u

segmentArcLength (self, segment, u0)

Description : In the segment find the arc length according to Eq. 43 for $u =$ u0.
Methods Called : SciPy.integrate.quad
Input Data:
- **segment :** (integer) the desired segment
- **u0 :** (scalar) the value of u for which arc length is desired

Return : (scalar) arc length measured from the beginning of the segment

cubicSolve (self, p, q, r, s)

Description : Solve the cubic equation $F = p\,u^3 + q\,u^2 + r\,u + s = 0,\ \ p, q, r, s \in \Re,$ and $0 \leq u \leq 1$. Multiple roots are possible.
Methods Called : cbrt
Input Data:
- **p :** (scalar) coefficient of u^3
- **q :** (scalar) coefficient of u^2
- **r :** (scalar) coefficient of u^1
- **s :** (scalar) coefficient of u^0

Return : (Boolean) True if roots are found; False otherwise, (integer) number of roots between 0 and 1, (list) u

3.2 Ray Class Methods

A Ray object is instantiated by passing a tuple of coordinates of its origin and a tuple with its direction in unit vector form (relative to the Ray origin). An example is shown below:

Example: Instantiation of Ray object.

```
>>> from WilsonFowler import Ray
>>> from math import sin, cos, radians
>>> # Assume: origin (x, y) = (−3., 4.5),
... direction (cos(θ), sin(θ)), θ = 80. degrees
>>> O = (-3., 4.5)
>>> theta = radians(80.)
>>> D = (cos(theta), sin(theta))
>>> ray = Ray(O, D)
>>> # Print Ray object
>>> print ray
Ray(Origin: (-3.0, 4.5), Direction: (0.173648177667, 0.984807753012))
```

Methods supporting instantiation of a Ray object are:

____init____(self, origin, direction)

Description : This is the constructor for the Ray-class. It contains the origin, direction, epsilon, and, if the ray has been cast against a spline, the results of the intersection of ray and WFS including the number of intersections found and the associated coordinate tuples, slopes, curvatures, radii from the origin, and unit normal and tangent tuples.
Methods Called : None
Input Data:
origin : (tuple) x and y coordinates of the ray's origin
direction : (tuple) x and y components of a unit vector parallel to the ray.

__str__(self)

Description : Return a string representation of the Ray-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinates of the ray origin and the ray direction components.

__repr__(self)

Description : Return a string representation of the Ray-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinates of the ray origin and the ray direction components.

Methods supporting ray-spline intersection are:

push (self, radius, xy, slope, curve, norm, tang)

Description : Append the input data to their respective lists when a ray-spline intersection has been found. Increment Crossings each time push is called.
Methods Called : None
Input Data:
- **radius :** (scalar) the distance from the ray origin to the point of intersection with the WFS
- **xy :** (tuple) the coordinates of the point of intersection with the WFS
- **slope :** (scalar) the slope of the WFS at the point of intersection with the WFS
- **curve :** (scalar) the curvature of the WFS at the point of intersection with the WFS
- **norm :** (tuple) the unit normal to the WFS at the point of intersection with the WFS
- **tang :** (tuple) the unit tangent to the WFS at the point of intersection with the WFS

Return : None

reset (self)

Description : Delete ray-spline intersection data, re-allocate memory, and reset the Crossings counter to 0
Methods Called : None
Input Data: None
Return : None

resetDirection (self, newDirection)

Description : Change the ray's direction, deleting any existing ray-WFS intersection data.
Methods Called : reset
Input Data:
- **newDirection:** (tuple) unit vector components for the ray direction

Return : None

3.3 Box Class Methods

A Box object is instantiated by passing a two tuples being the coordinates of the minimum x-y pair and maximum coordinate pair. An example is shown below:

Example: Instantiation of a Box object.

```
>>> from WilsonFowler import Box
>>> # Assume: min = (−3., 3.5), max = (−1., 4.5)
>>> Min = (-3., 3.5)
>>> Max = (-1., 4.5)
>>> box = Box(Min, Max)
>>> # Print Box object
>>> print segment
Box(min: (-3.0, 3.5), max: (-1.0, 4.5))
```

Methods supporting instantiation of a Box object are:

__init__(self, Min, Max)

Description : This is the constructor for the Box-class. The class contains a list of the input tuples.
Methods Called : None
Input Data:
  Min : (tuple) x and y coordinates of the Box's minimum coordinate pair
  Max : (tuple) x and y components of the Box's maximum coordinate pair

__str__(self)

Description : Return a string representation of the Box-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinates of the Box's minimum and maximum coordinate pairs.

__repr__(self)

Description : Return a string representation of the Box-class.
Methods Called : None
Input Data: None
Return : (string) the x,y coordinates of the Box's minimum and maximum coordinate pairs.

Methods supporting ray-segment intersection are:

RBintersect (self, ray)

Description : Determine if the input ray intersects the current box using the algorithm described on pp. 65-67 of An Introduction to Ray Tracing, Andrew S. Glassner, Ed., Morgan Cauffman, 1989.
Methods Called : sgn
Input Data:
  ray : (Ray object) The ray being cast against the box.
Return : (Boolean) True if the ray crosses the box, False otherwise.

Utility methods are:

sgn (self, A)

Description : Given a scalar, return its sign: $+1$ if $|A| \geq 0.$ or -1 otherwise.
Methods Called : None
Input Data:
  A : (scalar)
Return : ± 1

4 Example Problems

Error, accuracy, and other properties of the WFS are documented in, for example, Emery[3], Melvin[4], and Fritsch [6][7]. Here, several example problems are shown to illustrate the creation of a WFS and the various operations available on the spline which have not been covered in other articles. First, however, the sine function test case presented in the original article by Fowler and Wilson [1] is given.

4.1 The Sine-curve Test

Fowler and Wilson present a test of their spline calculations using the function $y = 2\sin(x)$, for $0 \leq x \leq 3\pi/2$. Tables 2 and 3 present a comparison of their results and those calculated here. For the case where slopes are specified (*i.e.*, in Table 2), the L_2-norm for differences in τ_a, τ_b, and strain energy are: $\|\Delta\tau_a\|_2 = 2.071 \times 10^{-4}$, $\|\Delta\tau_b\|_2 = 2.473 \times 10^{-4}$, and $\|\Delta\text{Energy}\|_2 = 3.991 \times 10^{-4}$. For the case with unspecified slopes (*i.e.*, in Table 3), the L_2-norm for differences in τ_a, τ_b, and strain energy are: $\|\Delta\tau_a\|_2 = 3.182 \times 10^{-4}$, $\|\Delta\tau_b\|_2 = 3.407 \times 10^{-4}$, and $\|\Delta\text{Energy}\|_2 = 1.724 \times 10^{-3}$. These measures suggest that the Python implementation for open curves has been done in a consistent manner, with differences likely due to computer architecture variations over the past 45 years. Note that the exact total strain energy for the sine-curve is 3.733877540.

As a test of the ray-spline intersection method, the results for the ray in Figure 3 are shown in Table 1. Agreement is shown to be at least to four decimal places.

Exact Intersection		Intersection with the WFS	
x	y	x	y
1.108835870	1.790360870	1.108803933	1.790395724
3.662686186	−0.99565765	3.662686666	−0.995658132
4.562881764	−1.977689197	4.562861934	−1.977667502

Table 1: Ray-spline intersection results for the ray shown in Figure 3. The intersection with the actual sine function are in the column labeled Exact Intersection, while those for intersection with the WFS for the original 28 points are shown in the last two columns.

4.2 A Non-Symmetric Closed Curve

A non-symmetric closed curve is shown in Figure 4 by the red polygon. Table 4 shows the data generated in the creation of the WFS. To test the ray-WFS intersection further, a ray is cast against the WFS in such a way that a total of four crossings occur, with two occurring within one segment, a test of the ability of the algorithm in getRayIntersection to handle multiple roots in a single segment.

5 Conclusions

A Python implementation of the Wilson-Fowler Spline algorithm has been developed and tested. Results show consistency between the current calculations and those reported by Fowler and Wilson [1] for the sine-curve test problem. The ability to place points between original WFS points (*i.e.*, point refinement), to distribute N-points along the WFS based upon M-points, and to cast rays against the WFS have been demonstrated. The Python implementation is found to be an effective and accurate means of computing and using Wilson-Fowler splines.

References

1. A. H. Fowler and C. W. Wilson. Cubic spline, A curve fitting routine. Report No. Y-1400 (Revision 1), Oak Ridge National Laboratory, June (1966).

Index	Fowler and Wilson [1],Table B-3					Results from WilsonFowler.py				
	Slope	τ_a	τ_b	Curvature	Energy	Slope	τ_a	τ_b	Curvature	Energy
1	2.00000	0.0020317	-0.0041206	0.0003	0.0001	2.00000	0.0020359	-0.0040958	0.0001	0.0001
2	1.96961	0.0083818	-0.0107739	-0.0316	0.0010	1.96971	0.0083976	-0.0107881	-0.0317	0.0010
3	1.87953	0.0157220	-0.0189114	-0.0694	0.0034	1.87949	0.0157130	-0.0188820	-0.0695	0.0034
4	1.73211	0.0249999	-0.0296377	-0.1216	0.0091	1.73222	0.0250363	-0.0296192	-0.1221	0.0091
5	1.53231	0.0379141	-0.0448701	-0.2046	0.0231	1.53234	0.0378990	-0.0449086	-0.2041	0.0232
6	1.28598	0.0566714	-0.0677822	-0.3416	0.0595	1.28594	0.0567040	-0.0677136	-0.3427	0.0595
7	1.00033	0.0847162	-0.1013267	-0.5900	0.1540	1.00038	0.0846912	-0.1014060	-0.5889	0.1541
8	0.684234	0.1235040	-0.1447909	-1.0171	0.3675	0.68419	0.1235179	-0.1447206	-1.0181	0.3673
9	0.346320	0.1624618	-0.1740647	-1.6378	0.6267	0.34635	0.1624617	-0.1740792	-1.6377	0.6268
10	0.146113e-07	0.1740647	-0.1624633	-2.0043	0.6267	0.00001	0.1741015	-0.1624544	-2.0051	0.6268
11	-0.346322	0.1447894	-0.1235031	-1.6387	0.3675	-0.34634	0.1447278	-0.1235427	-1.6372	0.3674
12	-0.684232	0.1013276	-0.0847166	-1.0171	0.1540	-0.68423	0.1013814	-0.0846896	-1.0182	0.1541
13	-1.00033	0.0677816	-0.0567134	-0.5901	0.0596	-1.00037	0.0677152	-0.0567008	-0.5892	0.0595
14	-1.28609	0.0448283	-0.0378810	-0.3418	0.0231	-1.28594	0.0449118	-0.0378983	-0.3428	0.0232
15	-1.53220	0.0296707	-0.0250905	-0.2044	0.0091	-1.53234	0.0296198	-0.0250358	-0.2041	0.0091
16	-1.73247	0.0188207	-0.0156811	-0.1216	0.0034	-1.73222	0.0188825	-0.0157139	-0.1221	0.0034
17	-1.87935	0.01078149	-0.0084539	-0.0695	0.0010	-1.87950	0.0107872	-0.0083940	-0.0695	0.0010
18	-1.96996	0.0040473	-0.0020243	-0.0312	0.0001	-1.96969	0.0040995	-0.0020497	-0.0316	0.0001
19	-1.99996	-0.0020231	0.0040476	-0.0000	0.0001	-2.00007	-0.0020497	0.0040994	-0.0000	0.0001
20	-1.96997	-0.0084549	0.0108151	0.0321	0.0010	-1.96969	-0.0083940	0.0107872	0.0316	0.0010
21	-1.87935	-0.0156808	0.0188205	0.0695	0.0034	-1.87950	-0.0157139	0.0188821	0.0695	0.0034
22	-1.73247	-0.0250908	0.0296709	0.1224	0.0091	-1.73222	-0.0250362	0.0296197	0.1221	0.0091
23	-1.53220	-0.0378809	0.0448281	0.2044	0.0231	-1.53234	-0.0378984	0.0449065	0.2041	0.0232
24	-1.28609	-0.0567134	0.0677821	0.3423	0.0596	-1.28595	-0.0567061	0.0677197	0.3427	0.0595
25	-1.00033	-0.0847162	0.1013267	0.5900	0.1540	-1.00036	-0.0846851	0.1013937	0.5889	0.1541
26	-0.684234	-0.1235040	0.1447909	1.0171	0.3675	-0.68421	-0.1235304	0.1447182	1.0184	0.3673
27	-0.346320	-0.1624618	0.1740647	1.6378	0.6267	-0.34635	-0.1624641	0.1740903	1.6376	0.6268
28	0.			2.0043		0.00000			2.0048	
			Total Energy:		3.7336			Total Energy:		3.7337

Table 2: Comparison of results between the current Python implementation and those reported in Table B-3 of [1]wherein the beginning slope was specified,to be $\mathbf{T}_1 = (1/\sqrt{5}, 2/\sqrt{5})$ and end slope was given as $\mathbf{T}_{28} = (1, 0)$. Data for node locations include slope, and curvature. The remaining are segment quantities.

2. SAMUEL M SELBY, ED.. Standard Mathematical Tables, 17th Edition, Chemical Rubber Company, Cleveland, OH, USA, (1969).
3. J.D. EMERY. Some properties of the Wilson-Folwler spline., Report BDX-613-2810, (DE82 020106), Bendix, Kansas City, MO, July (1982).
4. W. R.. MELVIN. Error analysis and uniqueness properties of the Wilson-Fowler spline, Report LA-9178 (Unlimited release), Los Alamos National Laboratory, August (1982).
5. SHARON K. FLETCHER. Recommended Practices for Spline Usage in CAD/CAM Systems, Report SAND84-0142 (Unlimited Release), Sandia National Laboratories, April (1984).
6. F.N FRITSCH. Energy comparisons of Wilson-Fowler splines with other interpolating splines, Report UCRL-93477, Rev. 1 (Unlimited release), Lawrence Livermore National Laboratory, January (1986).
7. F.N FRITSCH. History of the Wilson-Fowler spline, Report UCID-20746 (Unlimited release), Lawrence Livermore National Laboratory, April (1986).
8. F.N FRITSCH. Procedure for converting a Wilson-Fowler spline to a cubic B-spline with double knots, Report UCID-21325 (Unlimited release), Lawrence Livermore National Laboratory, October (1987).
9. RONALD M. DOLAN. The Wilson-Fowler spline in a global IGES coordinate frame, Report LA-11024-MS (Unlimited release), Los Alamos National Laboratory, September (1987).
10. ANDREW S. GLASSNER, ED..An Introduction to Ray Tracing, Morgan Kaufmann Publishers, Inc. (1989).

Index	Fowler and Wilson [1],Table B-4					Results from WilsonFowler.py				
	Slope	τ_a	τ_b	Curvature	Energy	Slope	τ_a	τ_b	Curvature	Energy
1	2.01444	0.0049028	-0.0049028	-0.0252	0.0002	2.01426	0.0048711	-0.0048711	-0.0251	0.0002
2	1.96580	0.0075996	-0.0105711	-0.0244	0.0009	1.96593	0.0076223	-0.0105856	-0.0246	0.0009
3	1.88045	0.0159248	-0.0189581	-0.0714	0.0034	1.88041	0.0159155	-0.0189341	-0.0714	0.0034
4	1.73192	0.0249531	-0.0296266	-0.1211	0.0091	1.73201	0.0249842	-0.0296055	-0.1216	0.0091
5	1.53234	0.0379251	-0.0448701	-0.2047	0.0231	1.53239	0.0379126	-0.0449105	-0.2043	0.0232
6	1.28598	0.0566714	-0.0677822	-0.3416	0.0595	1.28594	0.0567021	-0.0677085	-0.3427	0.0595
7	1.00033	0.0847162	-0.1013267	-0.5900	0.1540	1.00039	0.0846963	-0.1013923	-0.5891	0.1541
8	0.684234	0.1235040	-0.1447909	-1.0171	0.3675	0.68421	0.1235318	-0.1447308	-1.0183	0.3674
9	0.346320	0.1624618	-0.1740647	-1.6378	0.6267	0.34634	0.1624514	-0.1740903	-1.6373	0.6268
10	0.146113e-07	0.1740647	-0.1624633	-2.0043	0.6267	-0.00000	0.1740903	-0.1624514	-2.0049	0.6268
11	-0.346322	0.1447894	-0.1235031	-1.6387	0.3675	-0.34634	0.1447308	-0.1235320	-1.6373	0.3674
12	-0.684232	0.1013276	-0.0847166	-1.0171	0.1540	-0.68421	0.1013921	-0.0846955	-1.0183	0.1541
13	-1.00033	0.0677816	-0.0567134	-0.5901	0.0596	-1.00038	0.0677093	-0.0567054	-0.5891	0.0595
14	-1.28609	0.0448283	-0.0378810	-0.3418	0.0231	-1.28595	0.0449072	-0.0378994	-0.3427	0.0232
15	-1.53220	0.0296707	-0.0250905	-0.2044	0.0091	-1.53234	0.0296188	-0.0250362	-0.2041	0.0091
16	-1.73247	0.0188207	-0.0156811	-0.1216	0.0034	-1.73222	0.0188821	-0.0157140	-0.1221	0.0034
17	-1.87935	0.0108149	-0.0084539	-0.0695	0.0010	-1.87950	0.0107871	-0.0083941	-0.0695	0.0010
18	-1.96996	0.0040473	-0.0020243	-0.312	0.0001	-1.96969	0.0040994	-0.0020495	-0.0316	0.0001
19	-1.99996	-0.0020231	0.0040476	-0.0000	0.0001	-2.00007	-0.0020495	0.0040987	0.0000	0.0001
20	-1.96997	-0.0084549	0.0108151	0.0321	0.0010	-1.96970	-0.0083947	0.0107897	0.0316	0.0010
21	-1.87935	-0.0156808	0.0188205	0.0695	0.0034	-1.87948	-0.0157114	0.0188726	0.0695	0.0034
22	-1.73247	-0.0250908	0.0296964	0.1223	0.0091	-1.73226	-0.0250457	0.0296530	0.1220	0.0091
23	-1.53211	-0.0378554	0.0447309	0.2047	0.0230	-1.53223	-0.0378651	0.0447858	0.2045	0.0231
24	-1.28634	-0.0568107	0.0681854	0.3407	0.0601	-1.28627	-0.0568270	0.0681344	0.3413	0.0601
25	-0.999526	-0.0843119	0.0999105	0.5953	0.1507	-0.99954	-0.0842693	0.0999237	0.5945	0.1506
26	-0.686294	-0.1249276	0.1496064	0.9970	0.3865	-0.68635	-0.1250081	0.1497605	0.9971	0.3872
27	-0.341050	-0.1576277	0.1576277	1.7152	0.5497	-0.34083	-0.1574025	0.1574025	1.7129	0.5481
28	-0.0159980			1.7152		-0.01624			1.7129	
				Total Energy:	3.6728				Total Energy:	3.6719

Table 3: Comparison of results between the current Python implementation and those reported in Table B-4 of [1] wherein the beginning and ending slopes are not specified, but computed using Equations 25 and 26 with $C = 0$ in both cases. Data for node locations include slope, and curvature. The remaining are segment quantities.

11. W.H Press, S.A. Teukolsky, W.T. Vettering, and B.P. Flannery. Numerical Recipes in C: The Art of Scientific Computing. Second Edition, Cambridge University Press, New York (1992).

Index	Results from WilsonFowler.py				
	Slope	τ_a	τ_b	Curvature	Energy
1	-46.407540	-0.797465	0.434859	1.419692	1.720756
2	-0.502776	-0.315871	0.114878	0.142719	0.045246
3	-0.045568	-0.378485	0.852084	-0.026891	0.253122
4	1.635414	-1.560707	0.618130	0.202039	0.826349
5	-0.633935	-0.049490	0.276589	-0.102638	0.071348
6	-0.250773	-0.637386	0.585091	0.261565	0.367777
7	1.140943	0.065832	-0.748496	0.216673	0.262980
8	0.143840	-0.588446	1.3067458	-0.259655	4.043300
9	-46.407540			1.419710	
				Total Energy:	7.590878

Table 4: The WFS data for the non-symmetric closed curve of Figure 4. Data for node locations include slope, and curvature. The remaining are segment quantities.

Ray Intersection with the WFS	
x	y
-0.395427	2.283658
6.804357	8.043485
2.289137	4.431309
0.735261	3.188208

Table 5: Ray-spline intersection results for the ray shown for the non-symmetric closed curve in Figure 4. The ray origin is at $(-2., 1.)$ in direction $(0.780869, 0.624695)$.

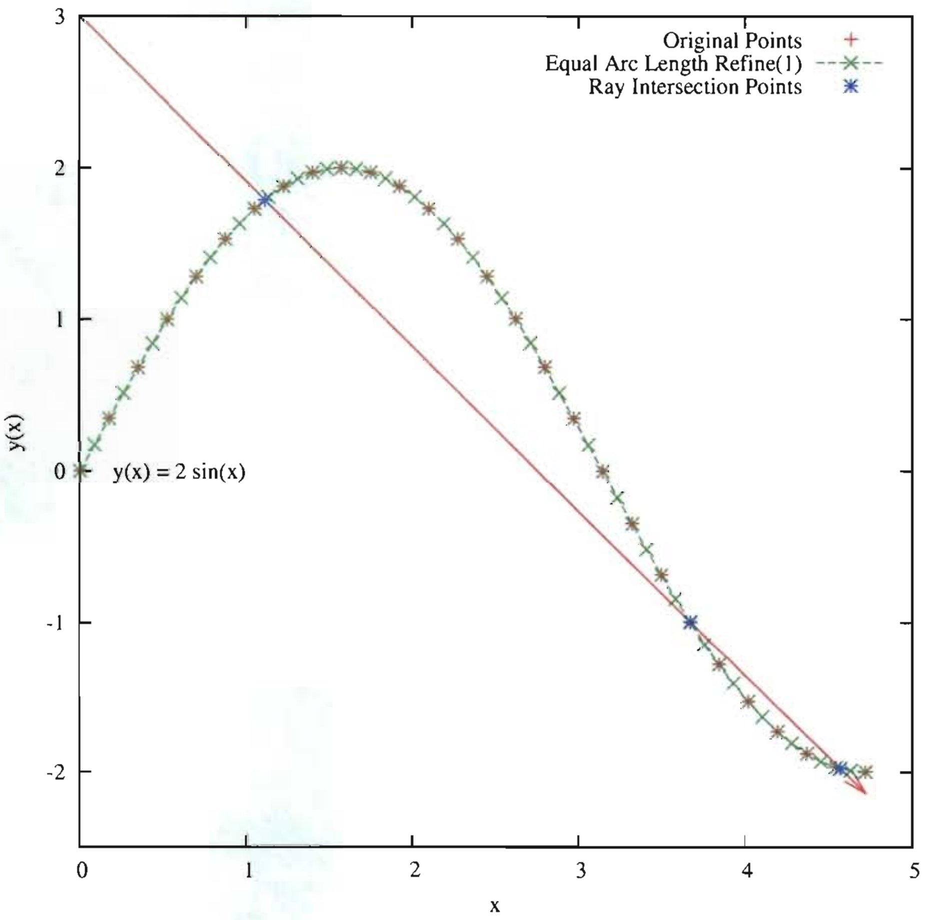

Fig. 3: The sine-curve test problem with: $y = 2\sin x$ and $x = 3\pi u/2$ for $0 \le u \le 1$. The original 28 points are in red and points generated by inserting 1 point between the original points using the equalArcLengthRefine method (end-point slopes specified) are in green. A ray originating at $(0, 3)$ in direction $(0.951056516295, -0.309016994375)$ intersects the spline at the three points listed in Table 1.

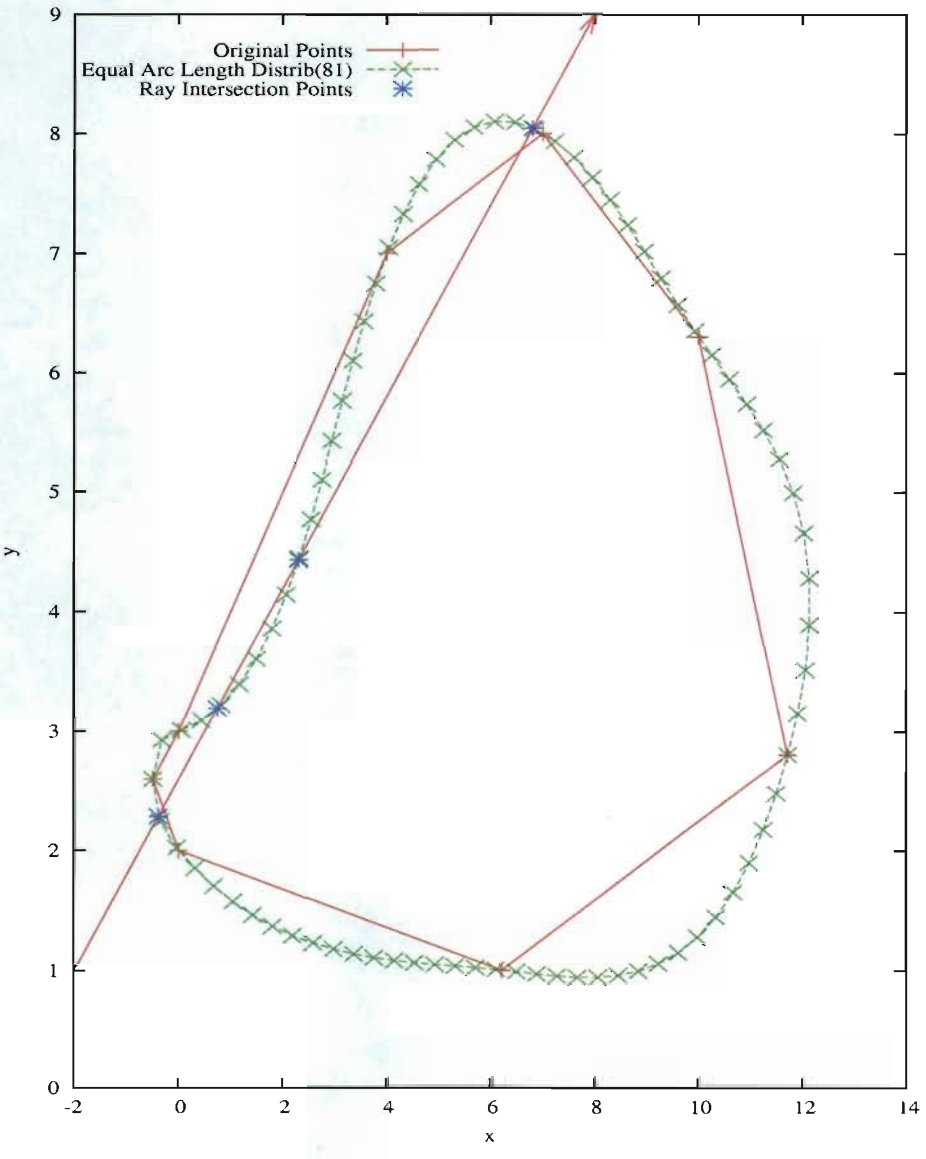

Fig. 4: A non-symmetric closed curve with 9 original points (red) and the results from an equal arc-length distribution of 81 points (green). The first and last points are at $(-0.5, 2.6)$ and the segments follow counterclockwise around the curve. A ray originating at $(-2, 1)$ in direction $(0.7808688, 0.624695)$ (red arrow) intersects the spline at the four (blue) points listed in Table 5. Notice that the ray is almost parallel to the segment having two ray-WFS crossings.