Chapter 1

Numerical Toolbox

One of the reasons for the enduring popularity of polynomial-based splines is the ease of computation, and straightforward numerical properties such as stability. By contrast, variational techniques are considered to be extremely slow and unwieldy. For example, Moreton's MVC solver took about 2.5 billion CPU cycles (75s on the hardware of the day) to compute the MVC solution to the 7-point data set posed by Woodford [10]. Even though more refined techniques have since been published, the prejudice continues. Kimia et al [4] state: "The major shortcomings of the elastica are that (i) they are not scale-invariant, (ii) they are computationally expensive to derive, and (iii) they do not lead to circular completion curves when the point-tangent pairs are, in fact, co-circular (i.e., both lie on the same circle)."

This chapter presents a toolbox of numerical techniques for computing sophisticated splines quickly, and with high precision. Section 1.1 presents a polynomial approximation to a fundamental integral that encompasses both the Euler spiral and the general polynomial spiral used as a primitive in Chapter **??**. Section 1.2 presents a very efficient solver fitting an Euler spiral segment to arbitrary endpoint tangent constraints. Section 1.3 presents a fast Newton solver for globally satisfying the continuity constraints for the interpolating spline.

*** TODO: integrate lookup table work ***

The combination of these techniques, along with optimized drawing code, which will be presented in Chapter **??**, yields an interpolating spline that is extremely efficient to compute, easily providing interactive response speeds. The complete spline is implemented and released in the open source libspiro library, which has also been integrated into Fontforge, a font editing tool, and Inkscape, a vector drawing editor.

The software release accompanying this thesis also includes a JavaScript implementation, which runs in standard Web browsers.

1.1 Numerical integration of polynomial spirals

Almost all calculations involving polynomial spiral curves can be reduced to an evaluation of a single definite integral:

$$spiro(k_0, k_1, k_2, k_3) = \int_{-.5}^{.5} e^{i(k_0 s + \frac{1}{2}k_1 s^2 + \frac{1}{6}k_2 s^3 + \frac{1}{24}k_3 s^4)} ds \tag{1.1}$$

Here, $k_0 ... k_3$ are the curvature and its first three derivatives at the center of a curve segment of arclength 1. More generally (for an arbitrary degree polynomial),

$$spiro(\mathbf{k}) = \int_{-.5}^{.5} e^{i \sum_{0 \le j} \frac{1}{(j+1)!} k_j s^{j+1}} ds. \tag{1.2}$$

Fixing the limits to $\pm.5$ simplifies the math and (as will be seen) creates symmetries which can be exploited to improve numerical accuracy, but does not lose generality. An integral over any limits can be expressed in terms of the spiro integral, as follows:

$$\begin{aligned}
&\int_{s_0}^{s_1} e^{i(k_0 s + \frac{1}{2}k_1 s^2 + \frac{1}{6}k_2 s^3 + \frac{1}{24}k_3 s^4)} ds = \\
&(s_1 - s_0) e^{i(k_0 s + \frac{1}{2}k_1 s^2 + \frac{1}{6}k_2 s^3 + \frac{1}{24}k_3 s^4)} \\
&\qquad spiro((s_1 - s_0)(k_0 + k_1 s + \tfrac{1}{2}k_2 s^2 + \tfrac{1}{6}k_3 s^3), \\
&\qquad\quad (s_1 - s_0)^2 (k_1 + k_2 s + \tfrac{1}{2}k_3 s^2), \\
&\qquad\quad (s_1 - s_0)^3 (k_2 + k_3 s), \\
&\qquad\quad (s_1 - s_0)^4 k_3), \\
&\text{where } s = \frac{s_0 + s_1}{2}
\end{aligned} \tag{1.3}$$

Note that, as a special case of Equation 1.3, the classic Fresnel integrals also reduce readily to this integral.

$$\int_0^s e^{it^2} dt = s e^{i\frac{s^2}{8}} spiro(\tfrac{s}{2}, s^2, 0, 0) \tag{1.4}$$

The spiro integral can be thought of as analogous to the sine and cosine functions of the generalized polynomial spiral. This connection is clear when considering a circular arc.

$$\frac{\sin x}{x} = spiro(2s, 0, 0, 0) \tag{1.5}$$

The absolute value of the spiro integral is the ratio of chord length to arc length of the curve segment (note that expressing this quantity as a ratio is invariant to scaling, and it is obviously invariant to rotation and translation as well). For a given curve segment with chord length Δx, the curvature and its derivatives along the curve (with arclength s normalized to the range $[-.5, .5]$) are given by a simple formula:

$$\kappa^{(j)}(s) = \Big(\frac{|spiro(\mathbf{k})|}{\Delta x}\Big)^{j+1} \sum_{0 \le i} \frac{k_{i+j}}{(i+1)!} s^i \tag{1.6}$$

1.1.1 Polynomial approximation of the integral

There are a number of approaches to numerically computing an integral such as Equation 1.1. However, a straightforward numerical integration technique such as Runge-Kutte would converge only relatively slowly, requiring significant computation time to achieve high accuracy. The approach of this section is to compute polynomial approximations of the integral, keeping all terms up to a certain order and discarding the rest.

A key insight is that the angle $\theta(s)$ is a straightforward polynomial function of s. Thus, to integrate $e^{i\theta(s)}$, take the Taylor's series expansion of e^x, substitute the polynomial $i\theta(s)$ for x, and expand symbolically, keeping terms only up to the given order.

$$\theta(s) = k_0 s + \frac{k_1}{2} s^2 + \frac{k_2}{6} s^3 + \frac{k_3}{24} s^4 \tag{1.7}$$

A worked example up to illustrate the process simpl

$$e^{ix} = 1 + ix - \frac{1}{2} x^2 \qquad + O(x^3) \tag{1.8}$$

Substituting $x = \theta(s)$, we get:

$$e^{i\theta(s)} = 1 + ik_0 s - \frac{ik_1 - k_0^2}{2} s^2 \qquad + O(s^3) \tag{1.9}$$

Integrating, we get:

$$\int e^{i\theta(s)} ds = f(s) = s + \frac{ik_0}{2} s^2 - \frac{ik_1 - k_0^2}{6} s^3 \qquad + O(s^4) \tag{1.10}$$

Evaluating $f(0.5) - f(-0.5)$, we find that all even polynomial terms (including s^4) drop out:

$$f(0.5) - f(-0.5) \approx 1 + \frac{ik_1 - k_0^2}{24} \tag{1.11}$$

Thus, as promised, fixing the limits of integration to ± 0.5 provides a symmetry that both decreases the number of terms to compute and also increases the accuracy by one degree.

Retaining all terms up to quartic yields an approximation not much more complicated. Note that this approximation actually uses all four parameters.

$$\Delta x \approx 1 + \frac{ik_1 - k_0^2}{24} + \frac{ik_3 - 6ik_0^2 k_1 + k_0^4 - 4k_0 k_2 - 3k_1^2}{1920} \tag{1.12}$$

1.1.2 Higher order polynomial approximations

Higher order polynomials offer a good way to increase the accuracy of the approximation at modest cost. However, manually working out the coefficients becomes quite tedious. This section describes a technique for automatically generating code for an arbitrary polynomial degree.

For functions of a single variable, the Taylor's series expansion gives the coefficients of the approximating polynomial simply and unambiguously (it is, of course, possible to use refinements such as Chebysheff polynomials to redistribute the errors more evenly, but the underlying concept remains). Here, the task is to approximate a function of four parameters. It is possible to generalize the Taylor's series expansion to functions of more than one parameter, but a simplistic approach would result in a blowup of coefficients; for a function of four parameters, the number of coefficients grows quadratically with the order of the polynomial.

A more sophisticated approach is represent each successive x^j as a vector of polynomial coefficients. Because of the symmetries of the equation (many odd terms will drop out), for $j \geq 3$, it is most efficient to compute x^j by combining the vectors for x^{j-2} and x^2. Then, these polynomials are substituted into the Taylor's series expansion for e^x. A fully worked out example, for order 8, is given below:

$$
\begin{aligned}
t_{11} &= k_0 \\
t_{12} &= \tfrac{k_1}{2} \\
t_{13} &= \tfrac{k_2}{6} \\
t_{14} &= \tfrac{k_3}{24} \\
t_{22} &= t_{11}^2 \\
t_{23} &= 2t_{11}t_{12} \\
t_{24} &= 2t_{11}t_{13} + t_{12}^2 \\
t_{25} &= 2(t_{11}t_{14} + t_{12}t_{13}) \\
t_{26} &= 2t_{12}t_{14} + t_{13}^2 \\
t_{27} &= 2t_{13}t_{14} \\
t_{28} &= t_{14}^2 \\
t_{34} &= t_{22}t_{12} + t_{23}t_{11} \\
t_{36} &= t_{22}t_{14} + t_{23}t_{13} + t_{24}t_{12} + t_{25}t_{11} \\
t_{38} &= t_{24}t_{14} + t_{25}t_{13} + t_{26}t_{12} + t_{27}t_{11} \\
t_{44} &= t_{22}^2 \\
t_{45} &= 2t_{22}t_{23} \\
t_{46} &= 2t_{22}t_{24} + t_{23}^2 \\
t_{47} &= 2(t_{22}t_{25} + t_{23}t_{24}) \\
t_{48} &= 2(t_{22}t_{26} + t_{23}t_{25}) + t_{24}^2 \\
t_{56} &= t_{44}t_{12} + t_{45}t_{11} \\
t_{58} &= t_{44}t_{14} + t_{45}t_{13} + t_{46}t_{12} + t_{47}t_{11} \\
t_{66} &= t_{44}t_{22} \\
t_{67} &= t_{44}t_{23} + t_{45}t_{22} \\
t_{68} &= t_{44}t_{24} + t_{45}t_{23} + t_{46}t_{22} \\
t_{78} &= t_{66}t_{12} + t_{67}t_{11} \\
t_{88} &= t_{66}t_{22} \\
u &= 1 - \tfrac{t_{22}}{24} - \tfrac{t_{24}}{160} - \tfrac{t_{26}}{896} - \tfrac{t_{28}}{4608} + \tfrac{t_{44}}{1920} + \tfrac{t_{46}}{10752} + \tfrac{t_{48}}{55296} - \tfrac{t_{66}}{322560} - \tfrac{t_{68}}{1658880} + \tfrac{t_{88}}{92897280} \\
v &= \tfrac{t_{12}}{12} + \tfrac{t_{14}}{80} - \tfrac{t_{34}}{480} - \tfrac{t_{36}}{2688} - \tfrac{t_{38}}{13824} + \tfrac{t_{56}}{53760} + \tfrac{t_{58}}{276480} - \tfrac{t_{78}}{11612160}
\end{aligned}
\tag{1.13}
$$

Obviously, for higher degrees, the formulas are too complex to be worked out by hand. The accompanying software distribution includes a utility, `numintsynth.py`, that synthesizes C code for an arbitrary degree. Asymptotically, the number of lines of code grows as $O(n^2)$, so for any reasonably low degree, the code is quite concise. Additionally, the structure of the dependency graph permits a high level of instruction-level parallelism, making the computation very efficient on typical CPUs.

1.1.3 Hybrid approach

The higher order the polynomial, the more accurate the result. However, code complexity grows significantly, and for small angles, accurate results are possible with a low-order polynomial.

A hybrid approach is to implement a polynomial of fixed order, and use it for parameter values where the error is within tolerance. For parameter values outside this tolerance, subdivide. The range $[-0.5, 0.5]$ is split into n equal subdivisions, each is evaluated using Equation 1.3, and the result is the sum of all such intervals.

The extreme case of an order-2 polynomial is equivalent to Euler integration. The code is simple, but convergence is expected to be poor.

The optimum order of polynomial is determined empirically. All even orders of polynomial between 2 and 16 were measured, and the accuracy plotted against CPU time. The results are shown in Figure 1.1.3, measured on a MacBook Pro with a 2.33GHz Core 2 Duo processor. Each successive datapoint doubles the total number of subdivisons. This particular plot is for evaluating $spiro(1, 2, 3, 4)$, but other combinations of arguments all give similar results.

The $n = 1$ case is relatively faster than $n > 1$, because no sine and cosine functions are needed for the subdivision. As can be seen, low order polynomials require a large number of subdivisions to provide accurate results. However, there is diminishing return as the order of the polynomial grows—while the number of intervals needed is decreased, the time needed to compute each one increases. As can be seen from the graph, orders greater than 12 don't improve overall performance significantly.

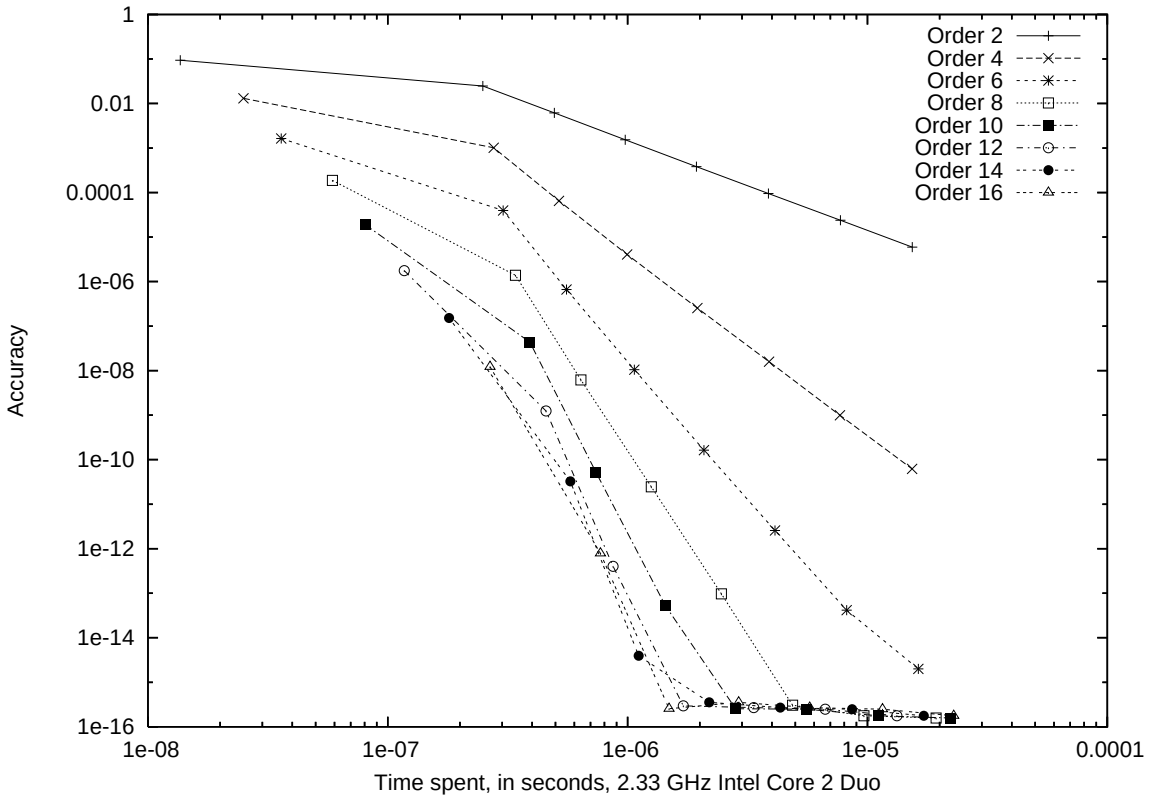

Figure 1.1: Precision vs timing for different integration strategies.

An empirically determined estimate for the error of the order 12 spiro approximation is:

$$|spiro_{12}(k_0, k_1, k_2, k_3) - spiro(k_0, k_1, k_2, k_3)| \approx (.006|k_0|^2 + .03|k_1| + .03|k_2|^{\frac{2}{3}} + .025|k_3|^{\frac{1}{2}})^6 \tag{1.14}$$

With n subdivisions, the error scales approximately as n^{-12}. Thus, it should be straightforward to compute a value for any given precision.

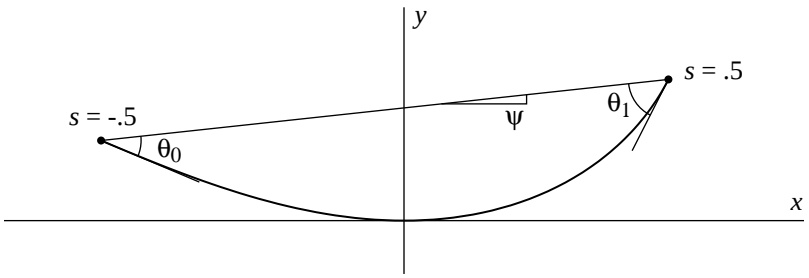

Figure 1.2: Determining Euler spiral parameters.

1.2 Determining Euler spiral parameters

One important primitive is to compute the parameters for Euler spiral segments, given a fixed chord length and tangents relative to the chord. This primitive is needed in the global determination of angles to satisfy the C2 curvature continuity constraints across knots, and is also independently of interest in vision applications, for computing a Euler spiral segment for use in shape completion.

There are quite a number of algorithms for this in the literature, the most detailed of which is given by Kimia et al[4]. Their solution represents the resulting curve in terms of two parameters:

$$\kappa(s) = \gamma s + \kappa 0, \text{ with } 0 \leq s \leq L \tag{1.15}$$

Numerically, their solution is complicated. They first construct a biarc to approximate the desired spiral segment, then use a gradient descent optimization method for the two parameters, computing the Euler spiral at each iteration.

Our solution is similar in many respects, but more direct; the biarc as an initial approximation is no longer needed. There are two key insights. First, by specifying the curvature of the curve from the middle, as opposed to one of the endpoints, one of the parameters has a closed-form solution. Second, it is easier to compute Euler spiral segments of fixed arclength than fixed chord length. Once that's done, scaling the resulting curve to fit the original chord length constraint does not affect the angle constraint.

We represent the curve to solve as:

$$\kappa(s) = k_0 + k_1 s, \text{ with } -.5 \leq s \leq .5 \tag{1.16}$$

Integrating, and fixing $\theta(0) = 0$,

$$\theta(s) = k_0 s + \frac{k_1}{2} s^2 \tag{1.17}$$

Again, the curve can be given in parametric form simply by integrating the angle:

$$x(s) + iy(s) = \int_0^s e^{i\theta(t)} \, \mathrm{d}t \tag{1.18}$$

We note that the endpoint angles with respect to the chord can be represented in terms of the angle of the chord in this coordinate frame:

$$\theta_0 = \psi - \theta(-.5) \tag{1.19}$$

$$\theta_1 = \theta(.5) - \psi, \text{ where} \tag{1.20}$$

$$\psi = \arctan \frac{y(.5) - y(-.5)}{x(.5) - x(-.5)} \tag{1.21}$$

But the formula for $\theta(s)$ is known (Equation 1.17), so:

$$\theta_0 = \frac{k_0}{2} - \frac{k_1}{8} + \psi \tag{1.22}$$

$$\theta_1 = \frac{k_0}{2} + \frac{k_1}{8} - \psi \tag{1.23}$$

From Equation 1.22 we can immediately solve for k_0:

$$k_0 = \theta_0 + \theta_1 \tag{1.24}$$

We have now reduced the search space to a single dimension, k_1. We will use a simple polynomial approximation to derive an initial guess, an approximate Newton-Raphson method to derive a second guess, and the secant method to further refine the approximation.

Using the small angle approximation $e^{i\theta} \approx 1 + i\theta$, Equation 1.18 becomes:

$$\begin{aligned} x(s) + iy(s) &\approx \int_0^s 1 + i\theta(t)\ \mathrm{d}t \\ &= s + i\Big(\frac{k_0}{2}s^2 + \frac{k_1}{6}s^3\Big) \end{aligned} \tag{1.25}$$

Combining Equations 1.21 and 1.25 (note the cancellation of even terms, just as for the integration in the previous subsection),

$$x(.5) - x(-.5) \approx 1 \tag{1.26}$$

$$y(.5) - y(-.5) \approx \frac{k_1}{24} \tag{1.27}$$

$$\psi \approx \arctan\frac{k_1}{24} \tag{1.28}$$

Applying the small angle assumption $\arctan\theta \approx \theta$ and substituting Equation 1.22,

$$\theta_0 \approx \frac{k_0}{2} - \frac{k_1}{12} \tag{1.29}$$

$$\theta_1 \approx \frac{k_0}{2} + \frac{k_1}{12} \tag{1.30}$$

Thus, we can derive a good approximation to k_1:

$$k_1 \approx 6(\theta_1 - \theta_0) \tag{1.31}$$

This approximation may be sufficient for low-precision applications. A considerably more accurate technique is to the secant method, which excels at finding the roots of nearly-linear one-dimensional functions. In this application, the function to be solved is angle error as a function of k_1, with θ_0 and θ_1 given.

The secant method needs two initial guesses. For the first, we choose $k_1 = 0$, simply because the angle error is trivial to calculate at that value. For the second guess, the approximation of equation 1.31 works well for $\theta_0 + \theta_1 < 6$ or so, but as this value approaches 2π, it significantly overestimates k_1 and sometimes fails to converge. The guess $6(\theta_1 - \theta_0)(1 - \frac{\theta_0+\theta_1}{2\pi})^3$ has been found to guarantee convergence for all $\theta_0 + \theta_1 < 2\pi$, and is also slightly more accurate, thus requiring fewer iterations to converge to a given precision.

Complete code for this method is appealingly concise; the Python version (shown in Figure 1.3) is only a dozen lines of code. Further, only a small handful of iterations is required to approach floating-point accuracy across the range, and a mere two or three for small angles. Given fast code for evaluating the spiro function, this approach is efficient indeed.

```
def fit_euler(th0, th1):
    k1_old = 0
    e_old = th1 - th0
    k0 = th0 + th1
    k1 = 6 * (1 - ((.5 / pi) * k0) ** 3) * e_old

    for i in range(10):
        x, y = spiro(k0, k1, 0, 0)
        e = (th1 - th0) + 2 * atan2(y, x) - .25 * k1
        if abs(e) < 1e-9: break
        k1_old, e_old, k1 = k1, e, k1 + (k1_old - k1) * e / (e - e_old)

    return k0, k1
```

Figure 1.3: Python code for computing Euler spiral parameters.

1.3 Global constraint solver

A common problem in computing interpolating splines is solving a global system of constraints. For example, in the framework of two-parameter extensible splines, the constraints are curvature continuity across control points. Other splines may demand a more complex set of constraints, such as the generalized four-parameter splines of Section **??**.

Most generally, the goal of the solver is to produce a vector of parameters, such that all the stated constraints are met. Most often, the relationship between the parameters and the corresponding constraints is *nearly linear*. For example, as demonstrated in Section 1.2, in the small angle approximation the relationship between endpoint tangent angles and curvature is given by linear Equations 1.29. Thus, one viable approach is to express the relationship between the parameters and constraints as a matrix, and solve it numerically. Hobby's spline [3] uses this approach, and achieves approximate G^2-continuity. The resulting matrix is tridiagonal, which admits a particularly fast $O(n)$ and simple solution.

However, this approach yields only an approximate solution, and as the bending angles increase, so do the curvature discontinuities in the resulting spline. A good general approach is to use Newton iteration on the entire vector parameters, using the Jacobian matrix of partial derivatives to refine each iteration. When a solution exists, convergence is quadratic, and experience shows that three or four iterations tend to be sufficient. This general technique resurfaces a number of times in the literature, including Malcolm [5], Stoer [9], and Edwards [1], the latter being an especially clear and general explication of using Newton-style iteration to solve nonlinear splines.

Newton iteration is one of the most efficient techniques, but far from the only workable approach. Since many of the splines are expressed in terms of minimizing an energy functional, a number of researchers use a gradient descent approach, such as Glass [2], and, more recently, Moreton [6]. However, gradient descent solvers tend to be much slower than Newton approaches. For example, Moreton's conjugate gradient solver for MVC curves took 137 iterations, and 75 seconds, to compute the optimal solution to a 7 point spline [6, p. 99]. By contrast, the techniques in this chapter easily achieve interactive speeds even in a JavaScript implementation that runs in standard Web browsers.

1.3.1 Two-parameter solver

For two-parameter splines, the solution is particularly easy. The vector of parameters is the tangent angle at each control point, and the constraints represent G^2-continuity across control points. Thus, the vector to be solved consists of the difference in curvature across each control point.

The vector of tangent angles is initialized to some reasonable values (in our implementation, the initial angles are those assigned by Séquin, Lee, and Yen's circle spline [8], although likely a simpler approach would suffice). At each iteration, the curvatures at the endpoints of each segment are computed (κ_i^l representing the curvature at the left endpoint of segment i, and κ_i^r the curvature at the right). The differences in curvature $\Delta\kappa_i = \kappa_i^r - \kappa_{i+1}^l$ form the error vector. At the same time, the Jacobian matrix represent the partial derivatives of the error vector with respect to the tangent angles:

$$J_{ij} = \frac{\partial \Delta\kappa_i}{\partial \theta_j} \tag{1.32}$$

Then, a correction vector $\mathbf{\Delta}\theta$ is obtained, by solving the Jacobian matrix for the given error vector. Since the Jacobian matrix is tridiagonal, the solution is very fast.

$$\mathbf{\Delta}\theta = \mathbf{J}^{-1}\mathbf{\Delta}\kappa \tag{1.33}$$

Finally, the θ_i values are updated for the next iteration. In all cases but the most extreme, the new value is just $\theta_i + \Delta\theta_i$. When bending angles are very large, this iteration can fail, and a second pass is attempted, using only half $\Delta\theta_i$ as the update; convergence is then only linear rather than quadratic, but it is more robust.

While each segment is specified by two parameters, this solver is written in terms of one parameter per control point, the tangent angle. This approach guarantees G^1-continuity by construction, as the same tangent is used on both sides of a control point.

One approach to determining the second parameter is to use an additional local solver, determining the two spiral parameters from the two tangent endpoints. For the Euler spiral spline, this local solver is described in Section 1.2. Another approach is to precompute a two-dimensional lookup table with the values.

In the software distribution, the clearest and most robust implementation of this solver is in `spiro.js`. The top-level Newton iteration is contained in `refine_euler`, the computation of the coefficients of the Jacobian matrix is in `get_jacobian_g2`, and the solution of the matrix uses the generic band diagonal solver `bandec` and `banbks`, adapted from Numerical Recipes [7].

1.4 Four-parameter solver

The same general approach is used for four-parameter splines, but the layout of parameters is a bit different. For one, the solver directly computes all parameters of each segment of the spline, rather than using a two-stage approach. This solver implements the general constraints as described in Section **??**.

The inner loop for computing the error vector is essentially a spiro integral (see Section 1.1). For the Jacobian matrix, for simplicity we use central differencing to compute the partial derivatives, even though an analytical technique would also be feasible.

Each constraint gets one element in the error vector and one row in the Jacobian matrix. For a constraint of the form, say $\kappa_2'' = \kappa_3''$, the entry in the error vector is the deviation from that equality, in this case $\kappa_2'' - \kappa_3''$. Each column in the Jacobian matrix represents one parameter of the polynomial spline, i.e. there are four times as many columns as curve segments. For the system to be well-determined, there must be an equal number of (linearly independent) constraints as parameters, i.e. the Jacobian matrix must be square. Thus, the constraints described in **??** are carefully designed to make the count come out even. In particular, for a sequence of G^2-continuous curve points, additional constraints zero out the higher-order derivatives of the polynomial spline, so that the resulting curve segments are segments of the Euler spiral.

The open-source libspiro package contains an efficient C implementation of this solver, the numerical methods being contained in `spiro.c`. The outer loop of the Newton iteration is contained in `spiro_iter`, with the computation of the Jacobian matrix in `compute_pderivs`, and the solution of the matrix in `bandec11` and `banbks11`, again adapted from Numerical Recipes, specialized to the size of the band.

An example of the four-parameter solver at work is shown in Figure 1.4. The first iteration (i.e. the linear approximation) is the thin red curve, and the second is in green. By the third iteration, the constraints are satisfied to within visual tolerance.

In practice, both solvers run efficiently and robustly. For the same problem, the two-parameter solver is a bit more robust than the four-parameter, but of course the latter is much more general and can solve a much wider class of spline problems.

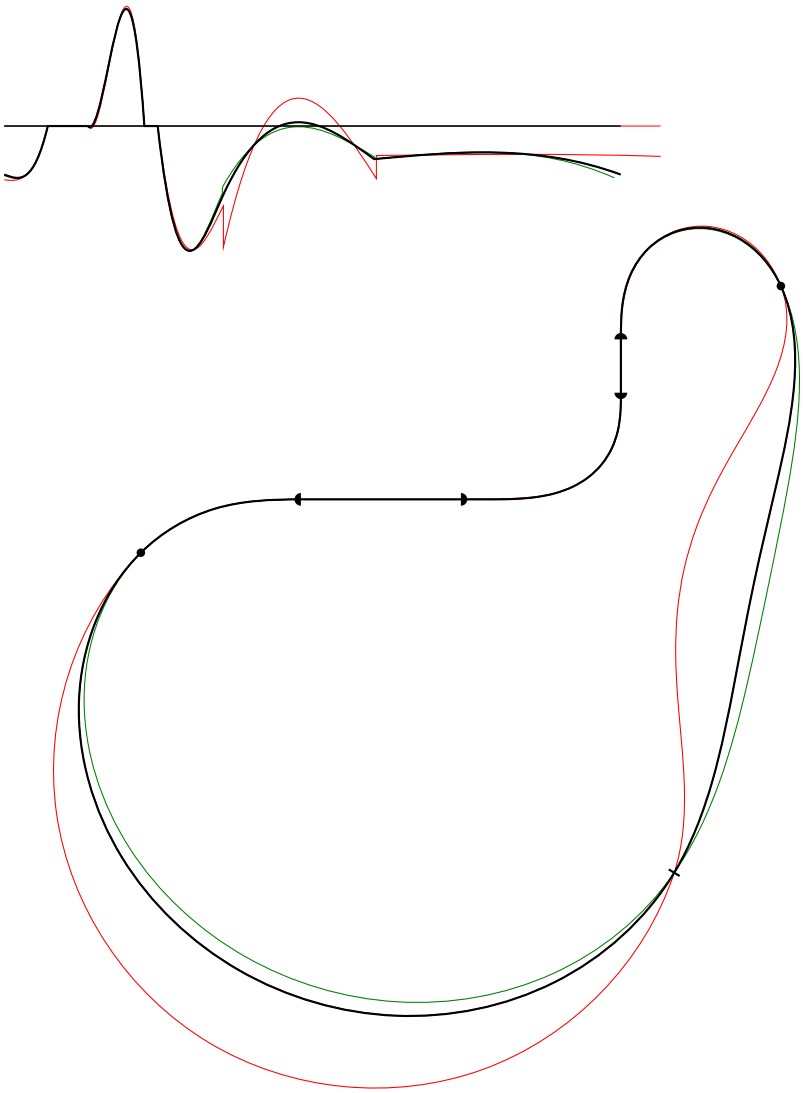

Figure 1.4: Example of four-parameter solver.

Bibliography

[1] John A. Edwards. Exact equations of the nonlinear spline. *Trans. Mathematical Software*, 18(2):174–192, June 1992.

[2] J. M. Glass. Smooth curve interpolation: A generalized spline-fit procedure. *BIT*, 6(4):277–293, 1966.

[3] John D. Hobby. Smooth, easy to compute interpolating splines. Technical Report STAN-CS-85-1047, Stanford University, 1985.

[4] Benjamin B. Kimia, Ilana Frankel, and Ana-Maria Popescu. Euler spiral for shape completion. *Int. J. Comput. Vision*, 54(1-3):157–180, 2003.

[5] Michael A. Malcolm. On the computation of nonlinear spline functions. *SIAM J. Numer. Anal.*, 14(2):254–282, April 1977.

[6] Henry Moreton. *Minimum Curvature Variation Curves, Networks and Surfaces for Fair Free-Form Shape Design*. PhD thesis, University of California, Berkeley, Berkeley, California, 1992.

[7] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 2nd edition, 1992.

[8] Carlo H. Séquin, Kiha Lee, and Jane Yen. Fair, G2- and C2-continuous circle splines for the interpolation of sparse data points. *Computer-Aided Design*, 37(2):201–211, 2005.

[9] Josef Stoer. Curve fitting with clothoidal splines. *J. Res. Nat. Bur. Standards*, 87(4):317–346, 1982.

[10] C. H. Woodford. Smooth curve interpolation. *BIT*, 9:69–77, 1969.