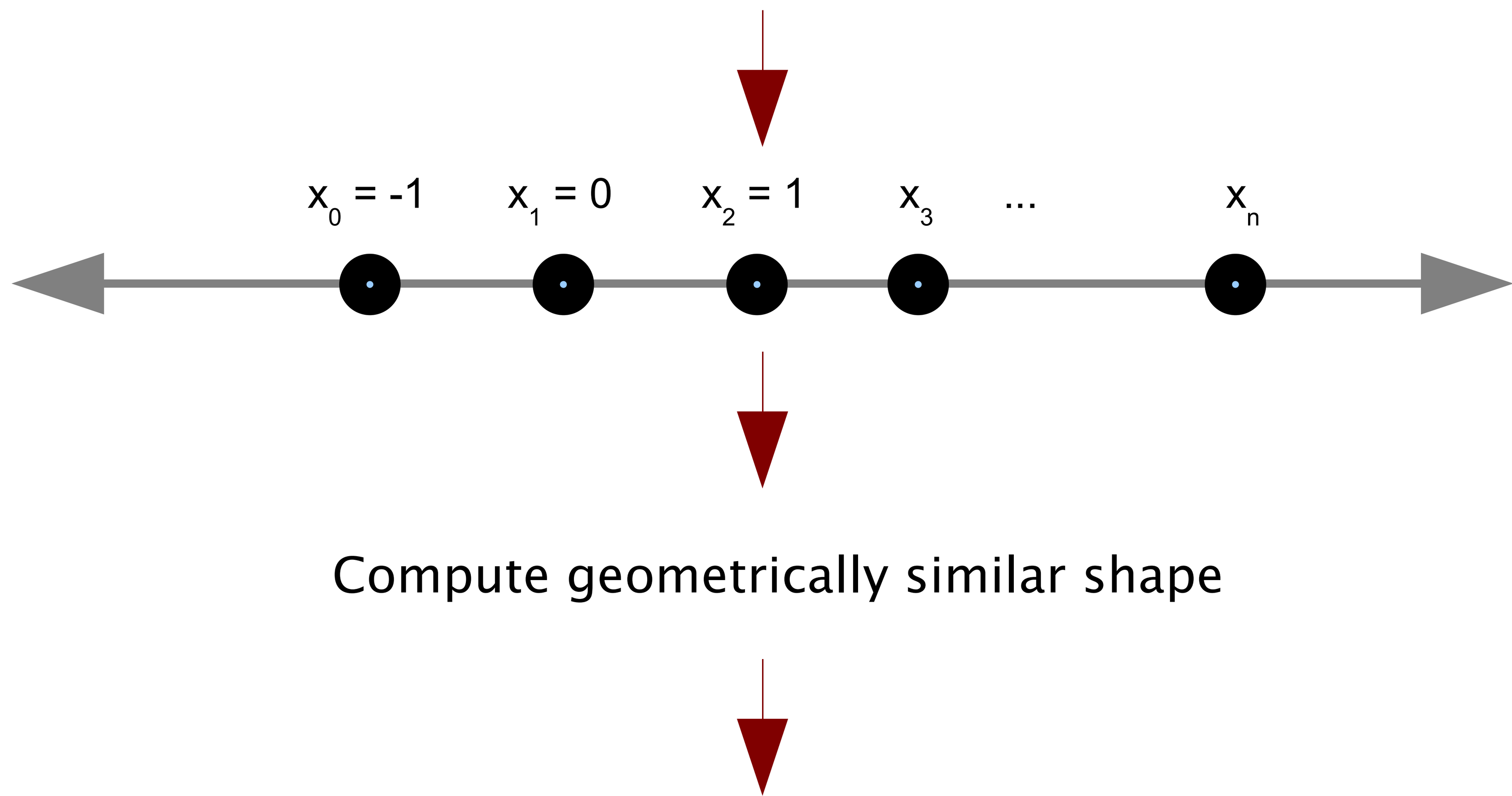
Conformal Mapping Using the Schwarz-Christoffel Transform

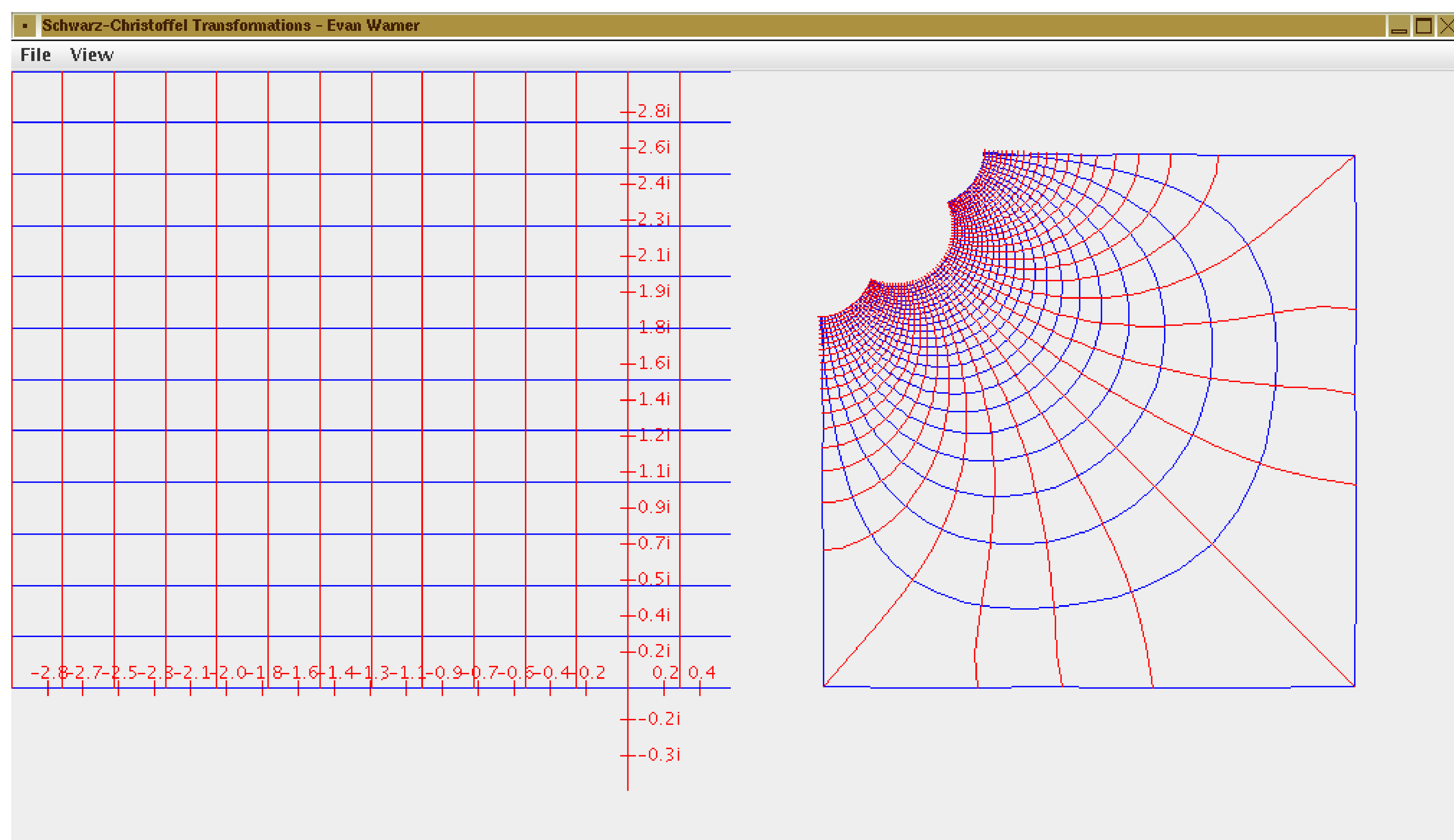
Evan Warner

Process

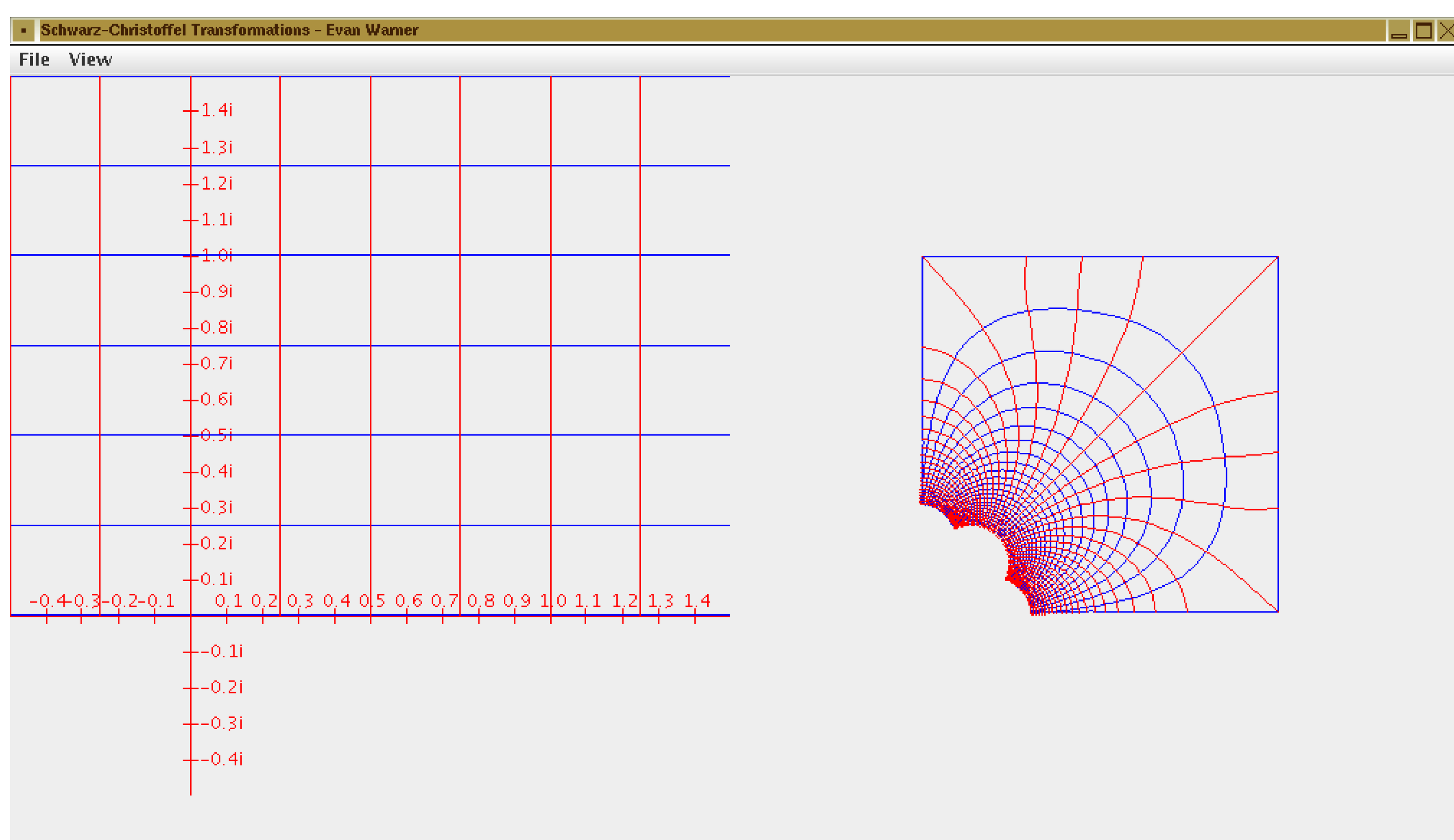
Calculate prevertices - Newton-Raphson method



Compute geometrically similar shape



Calculate auxiliary constants



Sample Code

```
public class ForwardGaussQuad
{
...
  public Complex recurse(int N, Complex a, Complex b, double al, double be)
  {
    if(a.imag()-b.imag()==0&&a.real()-b.real()==0)
      return new Complex(0,0);
    Complex half = a.add(b).divide(2.0);
    if(closeToSingularity(a,b,prevertex))
      return recurse(N, half, b, 0.0, be).add(recurse(N, a, half, al, 0.0));
    return integrateSubinterval(N, a, b, al, be);
  }
...
  public Complex integrateSubinterval(int N, Complex a, Complex b, double alpha, double beta)
  {
    GaussJacobiWeights gjw = new GaussJacobiWeights(0.0,beta,N);
    double[] points = gjw.getPoints();
    double[] weights = gjw.getWeights();
    double m = (b.real()+a.real())/2;
    double c = a.real()-m;
    double bimag = b.imag();
    Complex sum = new Complex(0.0,0.0);
    for(int i=0;i<weights.length;i++)
      sum=sum.add(value(new Complex(c*points[i]+m,bimag),-1,-1).multiply(weights[i]));
    sum=sum.multiply((new Complex(c,0)).power(beta+1.0));
    gjw = new GaussJacobiWeights(alpha,0.0,N);
    points = gjw.getPoints();
    weights  = gjw.getWeights();
    m = (b.imag()+a.imag())/2;
    c = a.imag()-m;
    double areal = a.real();
    Complex sum2 = new Complex(0.0,0.0);

    for(int i=0;i<weights.length;i++)
      if(alpha!=0.0)
        sum2=sum2.add(value(new Complex(areal,c*points[i]+m),1,-1).multiply(weights[i]));
      else
        sum2=sum2.add(value(new Complex(areal,c*points[i]+m),-1,-1).multiply(weights[i]));
    sum2=sum2.multiply(new Complex(0,c).power(alpha+1.0));
    if(a.imag()==0)
      sum2.imag(-sum2.imag());
    return sum.add(sum2);
  }
...
}

public class GaussQuad
{
...
  public double integrate(int N)
  {
    GaussJacobiWeights gjw = new GaussJacobiWeights(alpha,beta,N);
    double[] points = gjw.getPoints();
    double[] weights = gjw.getWeights();
    double m= (b+a)/2;
    double c= b-m;
    double sum = 0;
    for(int i=0;i<weights.length;i++)
      sum+=f.value(c*points[i]+m)*weights[i];
    sum=sum*Math.pow(c,alpha+beta+1);
    return sum;
  }
}

public class RealNewtonRaphson
{
...
  private double[] solvematrix(double[][] A, double[] b)
  {
    int l=n-3;
    double[][] L = new double[l][l];
    double[] q = new double[l];
    double[] x = new double[l];
    for(int p=0;p<l;p++)
      L[p][p]=1;
    for(int p=0;p<l;p++)
      for(int r=p+1;r<l;r++)
      {
        for(int c=p+1;c<l;c++)
        {
          A[r][p]=0;
          A[r][c]=A[r][c]-L[r][p]*A[p][c];
        }
      }
    for(int r=0;r<l;r++)

    {
      double sum=0;
      for(int c=0;c<r;c++)
        sum+=L[r][c]*q[c];
      q[r]=(b[r]-sum);
    }
    for(int r=0;r<l;r++)
    {
      double sum=0;
      for(int c=l-r;c<l;c++)
        sum+=A[l-r-1][c]*x[c];
      x[l-r-1]=(q[l-r-1]-sum)/A[l-r-1][l-r-1];
    }
    return x;
  }
  private double[][] jacobian(double[] funcvalues, double[] x)
  {
    double[][] jac=new double[n][n];
    for(int j=0;j<(n-3);j++)
    {
      double temp = x[j+2];
      double h = TOL*Math.abs(temp);
      if(h==0)
        h=TOL;
      x[j+2]=temp+h;   //Reduces floating-point error
      h=x[j+2]-temp;
      double[] funcvalues2=function(x);
      x[j+2]=temp;
      for(int i=0;i<(n-3);i++)
        jac[i][j]=(funcvalues2[i]-funcvalues[i])/h;
    }
    return jac;
  }
...
  private double[] function(double[] x)
  {
    x=xhattox(x);
    double[] xwithstuff=new double[n-1];
    xwithstuff[0]=-1;
    xwithstuff[1]=0;
    for(int i=2;i<(n-1);i++)
      xwithstuff[i]=x[i-2];
    double[] f = new double[n-3];
    SchwarzFunction denom = new SchwarzFunction(x,angle,0);
    GaussQuad gq = new GaussQuad(denom, denom.alpha(), denom.beta(), denom.a(), denom.b());
    double den = Math.abs(gq.integrate(GQN));
    double den1 = vertex[1].subtract(vertex[0]).modulus();
    for(int i=0;i<(n-3);i++)
    {
      SchwarzFunction numer = new SchwarzFunction(x,angle,i+1);
      gq = new GaussQuad(numer, numer.alpha(), numer.beta(), numer.a(), numer.b());
      double num = Math.abs(gq.integrate(GQN));
      double num1 = vertex[i+2].subtract(vertex[i+1]).modulus();
      f[i]=num/den-num1/den1;
    }
    return f;
  }
```

References

[1] Howell, L. H. (1990). *Computation of conformal maps by modified Schwarz-Christoffel transformations*. Retrieved September 28, 2007, from http://citeseer.ist.psu.edu/howell90computation.html.

[2] Saff, E. B., & Snider, A. D. (n.d.). *Funamentals of complex analysis with applications to engineering, science, and mathematics*. Prentice-Hall Engineering/Science/Mathematics.

[3] Trefethen, L. (1979). *Numerical computation of the Schwarz-Christoffel transformation*. Retrieved September 28, 2007, from ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/79/710/CS-TR-79-710.pdf.

[4] Press W., Teukolsky, S., Vetterling, W., & Flannery, B. (1992). *Numerical Recipes in C, Second Edition*. Cambridge University Press.

[5] O'Connor, J.J., & Robertson, E.F. (2001). *Hermann Amandus Schwarz*. Retrieved November 3, 2007, from http://www-history.mcs.st-andrews.ac.uk/Biographies/Schwarz.html.

Introduction and Background

Many physical problems are expressed as differential or boundary value problems over a surface. Often, these surfaces are or can be approximated by two-dimensional polygons. In this specific case, one method of determining accurate solutions is by taking the polygonal domain to exist in the complex plane and determining a conformal map, which preserves the structure of Laplace's equation, that restates the problem in a simpler domain, most often the upper half-plane. The new problem, now easy to solve analytically or in closed form, is then mapped back to the original domain. For such polygonal domains, a method of determining the specific transform needed is provided by the following formula, known as the Schwarz-Christoffel transform:

$$f(z) = A \int_0^z \prod_{j=1}^{n} (\zeta - x_j)^{-\theta_j/\pi} d\zeta + B. \tag{1}$$

In this formula, ζ is an independent complex variable in the upper half-plane, the θ_j are the exterior angles of the polygon, the x_j are 'preverticies' of the mapping (given along the real axis), n is the number of verticies of the polygon, and A and B are complex constants that specify the location of the image polygon in the complex plane. The θ_j must satisfy

$$\sum_{j=1}^{n} \theta_j = 2\pi, \tag{2}$$

which ensures the completeness of the image polygon [2]. Unfortunately, the Schwarz-Christoffel formula is not easy to evaluate, and requires both effective integration algorithms and efficient, convergent nonlinear equation solvers. Implementation of such numerical routines is not a trivial problem, and is the goal of this project.

The first problem in calculating the Schwarz-Christoffel mapping is the evaluation of the integral given by Eq. (1). The integrand contains singularities at each of the endpoints of the image polygon, which tend to render ordinary numerical integration routines either useless or hopelessly slow. In addition, the presence of negative powers in f means that domains of applicability for each of the subfunctions $(\zeta - x_j)^{-\theta_j/\pi}$ must be chosen so that the entire domain in and immediately around the image polygon is meromorphic. Although several quadrature routines have been used for this problem, the method of choice today is Gauss-Jacobi quadrature, which uses a specially-tailored weighting function to choose points of evaluation and weights for the points that maximize efficiency. In practice, the Schwarz-Christoffel formula is altered so that the prevertex x_n is chosen to be both $-\infty$ and $+\infty$ (the values are equivalent for a conformal map, which acts on the Riemann sphere). This can always be done due to the extra degrees of freedom contained in Eq. (1). The integrals that must be evaluated in practice in the course of the Schwarz-Christoffel transform are of the form

$$\int_{x_{i-1}}^{x_i} \prod_{j=1}^{n-1} (\zeta - x_j)^{-\theta_j/\pi} d\zeta. \tag{3}$$

These integrals can always be written as required for Gauss-Jacobi quadrature; that is, in the form

$$\int_a^b (z-a)^{\alpha} (z-b)^{\beta} \psi(z) dz, \tag{4}$$

where α and β are real numbers greater than -1.

The points and weights of a Gauss-Jacobi quadrature are calculated here using a routine from Numerical Recipes [4] which efficiently estimates and solves for the roots of the Jacobi polynomials, which form the sample points just as the roots of the Chebyshev polynomials form the sample points for standard Gaussian quadrature. These points, however, are uniformly calculated in the range $[-1, 1]$, and the integrals must be adjusted slightly to conform to this range. During the calculation of the preverticies, discussed below, the z in Eq. (4) will be restricted to the real axis; however, in direct calculations once the preverticies have been found, the z will generally be fully complex, which must be dealt with by the program.

The second problem is the Schwarz-Christoffel parameter problem, where the x_j in Eq. (1) are calculated. As described in [2], a series of nonlinear, constrained equations can be formed from the requirement that the image polygon and the desired polygon be similar (the constants A and B in Eq. (1) then ensure congruency). Written out, there are $n-3$ linear equations in $n-3$ unknowns, once the extra degrees of freedom have been taken care of by arbitrarily giving three of the x_j precise values. Here, as in the literature, we take $x_1 = -1$ and $x_2 = 0$ in addition to the already-defined $x_n = \pm\infty$. The equations to be solved are then

$$\frac{|\int_{x_{i-1}}^{x_i} \prod_{j=1}^{n} (\zeta - x_j)^{-\theta_j/\pi} d\zeta|}{|\int_{x_1}^{x_2} \prod_{j=1}^{n} (\zeta - x_j)^{-\theta_j/\pi} d\zeta|} - \frac{|w_j - w_{j-1}|}{|w_2 - w_1|} = 0, \tag{5}$$

where $i = 3, 4, ..., n-1$. However, there is an additional complication, as the order of the preverticies on the real axis matters. The extra constraint can be expressed as

$$0 < x_3 < x_4 < \ldots < x_{n-1} < \infty. \tag{6}$$

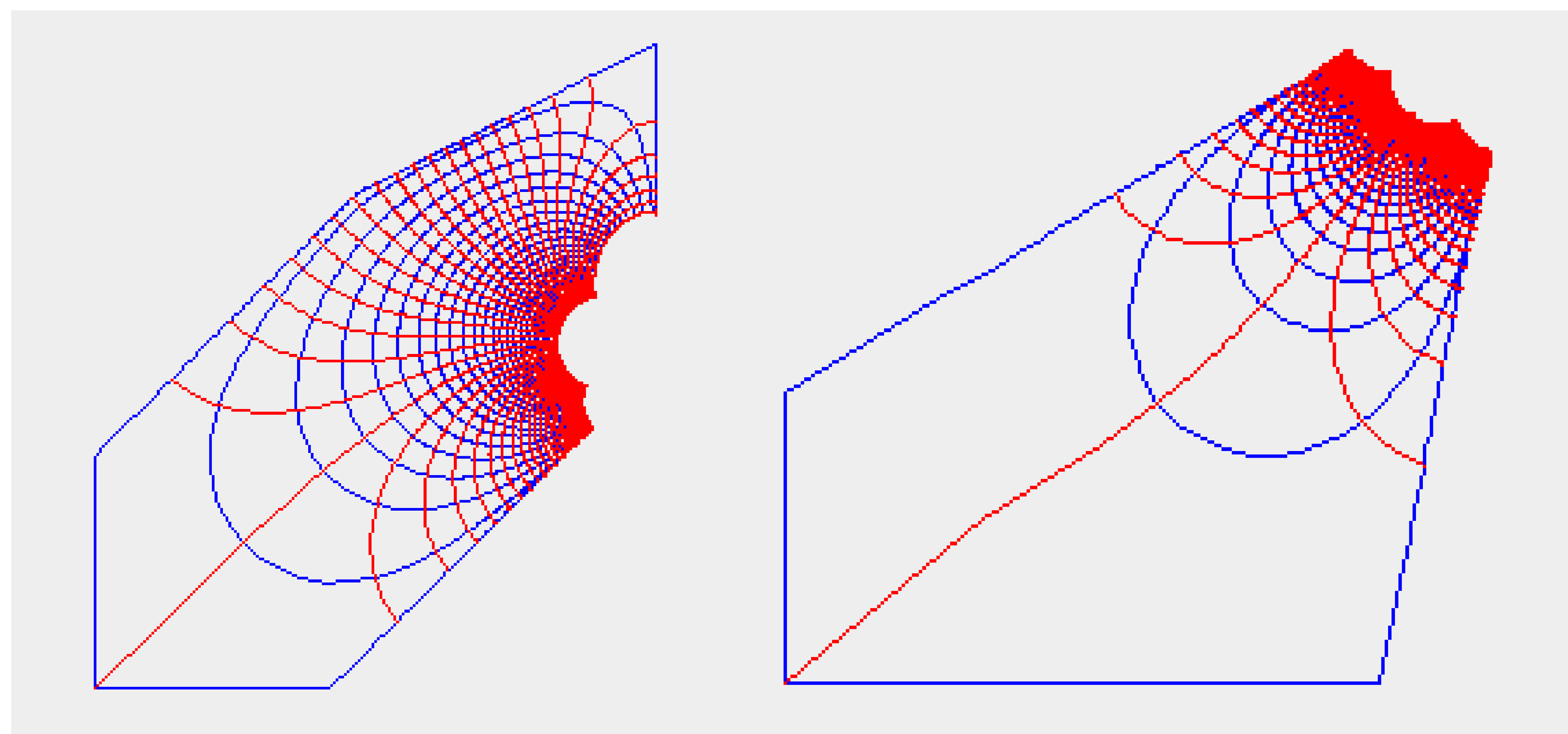
The nonconstrained problem is relatively easy to solve; however, the constraint prevents a naive application of a Newton's Method variant to this problem. To get around this, Trefethen in [3] suggests a simple change of variables that ensures the inequalities of Eq. (6). Take a new series of variables, χ_j, and let

$$\chi_j = \ln(x_j - x_{j-1}). \tag{7}$$

The resulting χ_j will automatically obey Eq. (6), and the original x_j are found by the simple inverse formula

$$x_j = x_{j-1} + e^{\chi_j}. \tag{8}$$

This new set of equations in the χ_j is readily solved by a variant of Newton's Method that does not require an explicit calculation of the Jacobian matrix (which would be hopelessly complex), but rather uses progressive estimates.



Example mappings produced by the program

Development

The software is written entirely in Java, with testing in MATLAB. The entire development of the program is designed to be achieved in stages by attacking the subproblems individually. Several important classes are described below:

- class Complex - this class stores and performs arithmetic on complex numbers, which are not directly supported by Java. Several of the methods, including the multiplication and division algorithms, are designed to run as quickly as possible while avoiding intermediate overflow and floating-point error propagation. The multiplication method, for instance, requires only three real multiplications rather than four.
- class GaussJacobiWeights - this class calculates and stores the sample points and weights for a given Gauss-Jacobi quadrature over the interval $[-1, 1]$. This routine uses Newton's Method to find the roots of the Jacobi polynomials, which are the sample points for the integral, and was taken and translated from [4].
- class SchwarzFunction - this class evaluates the integrand of a given real-valued Schwarz-Christoffel integral, serving as a storage class for data of this kind.
- class GaussQuad - this class accepts as input ψ, a, b, α, and β from Eq. (4). For an arbitrary integral in that form, shifting and scaling the bounds produces the equivalent integral

$$c^{\alpha+\beta+1} \int_{-1}^{1} (\zeta - 1)^{\alpha} (\zeta + 1)^{\beta} \psi(c\zeta + m) d\zeta, \tag{9}$$

  where $b = \frac{a+b}{2}$ and $c = b - m = m - a$. This integral is then evaluated using the sample points and weights given by the GaussJacobiWeights class and returned. For any GaussQuad object, varying numbers of sample points (and thus varying accuracy) are accepted by its integrate() method.
- class RealNewtonRaphson - this class accepts an array of vertices and calculates the necessary prevertices as well as the constants A and B from Eq. (1). The method employs a standard Newton-Raphson method to solve the Eq. (5). At each step, an approximate Jacobian matrix for the function is calculated using a forward-difference method in each dimension; the step vector is then solved for using an LU factorization on the equation

$$\mathbf{J}\delta\vec{x} = \vec{f}, \tag{10}$$

  where $\mathbf{J}$ represents the Jacobian, $\delta\vec{x}$ the step vector, and $\vec{f}$ the current function vector. Note that by employing a forward-difference method to find the Jacobian, the number of function evaluations can be cut in half, as the current function vector can be reused in the Jacobian calculation.
- class ForwardGaussQuad - this class, using already-calculated values for the prevertices, evaluates the Schwarz-Christoffel integral at a given point. To minimize error caused by the presence of singularities near the path of the integral (the singularities at the endpoints are handled by the Gauss-Jacobi quadrature), the path of integration is divided recursively such that no segment is closer to a singularity than one-half its length, a technique employed in [3]. Such recursive subdivision is known as compound Gauss-Jacobi quadrature.
- class SchwarzChristoffel - this class runs the graphical user interface and calls RealNewtonRaphson and ForwardGaussQuad when necessary. The graph itself has the ability to show axes and manually adjust window parameters.

In future iterations of the project, a new set of routines will be implemented to calculate continuous Schwarz-Christoffel problems. Immediately following from Eq. (3) above, we have

$$f'(z) = A \prod_{j=1}^{n-1} (\zeta - x_j)^{-\theta_j/\pi}. \tag{11}$$

To change this into a continuous problem, we can rewrite this as

$$f'(z) = A e^{\frac{1}{\pi} \sum_{j=1}^{n-1} -\theta_j \ln(z - x_j)}. \tag{12}$$

Then, defining the natural logarithm function as single-valued in the upper half-plane, except where $x_i = z$, f' becomes an analytic function in the required domain. To formulate the continuous-boundary problem, we simply replce the sum in Eq. (12) with an integral, and integrate the entire function to find $f(z)$:

$$f(z) = A \int_0^z e^{\frac{1}{\pi} \int_{-\infty}^{\infty} -\theta(x) \ln(\zeta - x_j) dx} d\zeta + B, \tag{13}$$

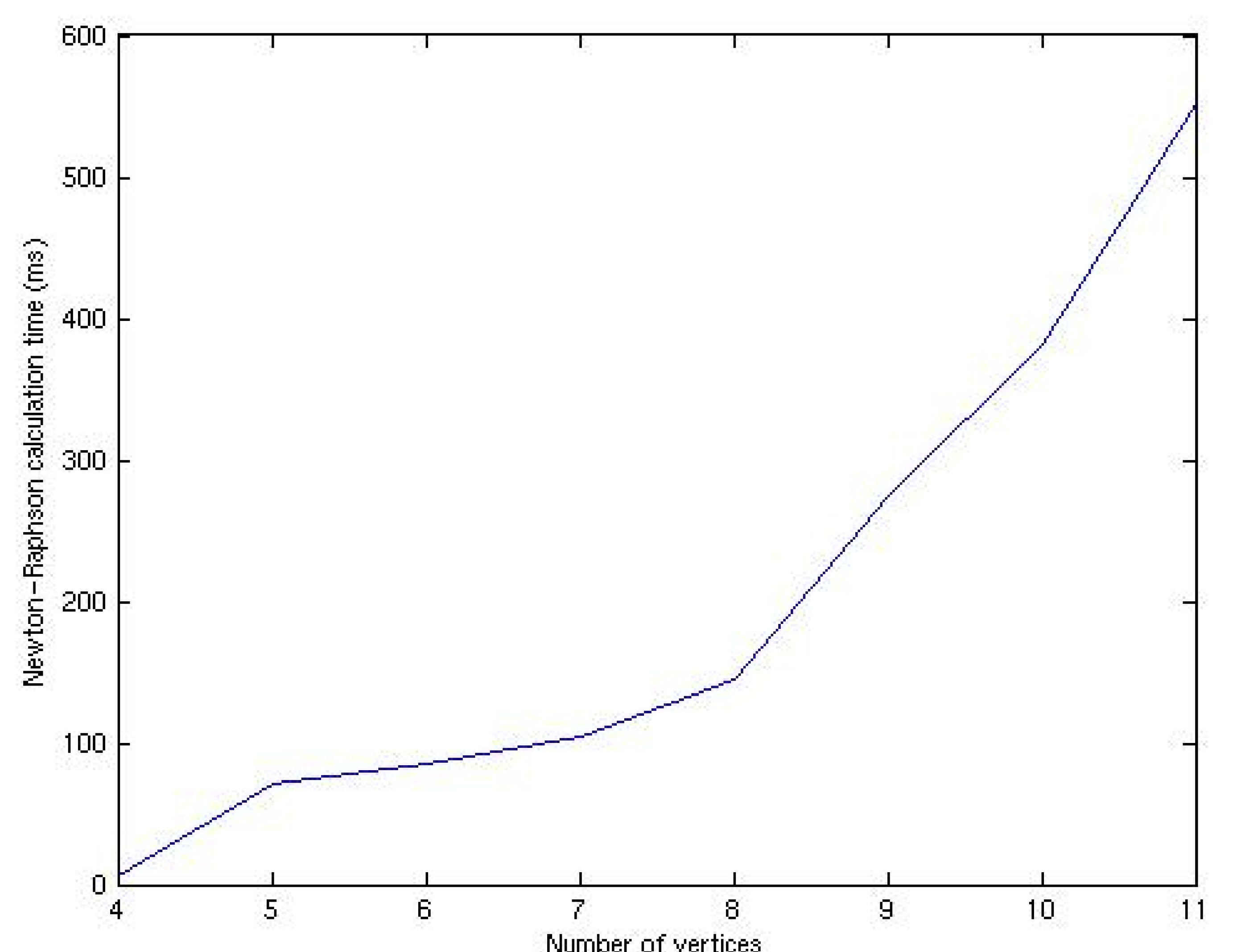
where $\theta(x)$ represents the amount of turning per unit length on the real axis, such that

$$\int_{-\infty}^{\infty} \theta(x) dx = 2\pi. \tag{14}$$

The continuous problem therefore has an extra subproblem to solve, namely, the solution of the integral equation, Eq. (13), to find $\theta(x)$ at every x.

Results

Results on polygons eleven and fewer vertices suggest that the program has both a reasonable error tolerance and reasonable computation time requirements. Efficiency has been gained by implementing a two-step integration in ForwardGaussQuad which deals with the real and complex parts of the path integral separately and minimizes the average depth of recursion.



Runtime data for a typical computer