Adaptive Interior Laplace BVP

Dan Fontaine

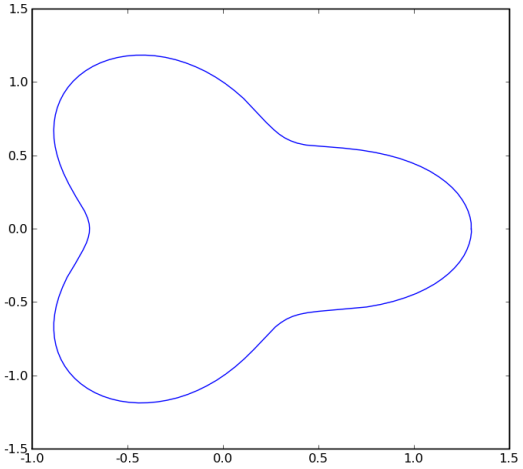
December 10, 2008

1 Laplace Boundary Value Problem

In this paper I will apply adaptive quadrature to solve an interior Dirichlet boundary value problem. We will solve the double-layer operator

$$(D\tau)(x) := \int_{\partial\Omega} \frac{\partial\Phi(x,y)}{\partial n_n}\tau(y)ds_y$$

where $\Phi(x,y) = -1/(2\pi)ln|x-y|$ is the fundamental solution for the Laplace equation. Our domain Ω will be defined to be the interior of the curve defined by $y(t) = (R(t)\cos t, R(t)\sin t), R(t) = 1 + .3\cos 3t$.



We will use a boundary value problem with known solution $u(x,y) = cos(x)e^y$, so our Dirichlet boundry data is $f = u|_{\partial\Omega}$. To solve this problem at a point x in the interior of Ω we integrate over $\partial\Omega$ using periodic quadrature

with equal node spacing. But we first must solve for the density τ at the nodes using the 2nd-order Fredholm equation

$$(I - 2D)\tau = -2f$$

We do this by using the Nystom method. We construct the Nystrom matrix using the following python code:

```
def nystromMatrix(T, W, y, dy, ddy, ny):
        k = curvature(T, dy, ddy, ny)

        lengthOfT = len(T)
        D = zeros((lengthOfT, lengthOfT))
        for i in range(0, lengthOfT):
                for j in range(0, lengthOfT):
                        speed = sqrt(dot(dy[j], dy[j]))
                        if i == j:
                                D[j, j] = -k[j] * W[j]
                                   * speed / (4. * pi)
                        else:
                                D[i, j] = fundSolDeriv(
                                   y[i], y[j], ny[j]) *
                                    W[j] * speed
        return D
```

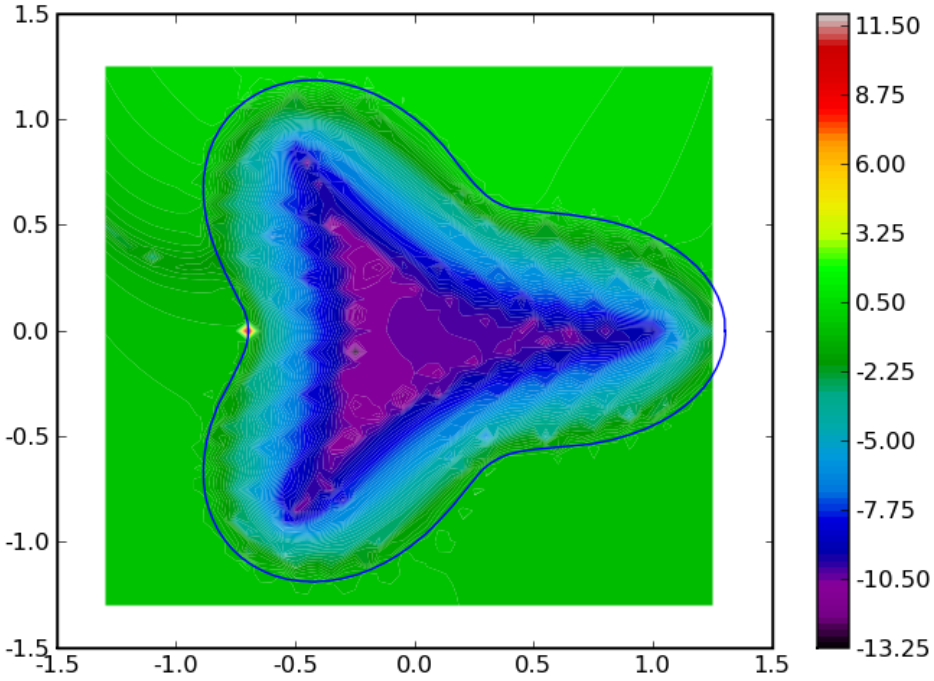
Notice that computing $\frac{\partial \Phi(x,y)}{\partial n_n}$ along the diagnol produces a singularity, so we fill the diaganol by computing the curvature $\kappa(t)$ for $t \in [0, 2\pi)$ corresponding to the position of our nodes. Here is the code for computing curvature:

```
def curvature(t, dy, ddy, ny):
        num = zeros(size(t))
        den = zeros(size(t))
        for j in range(0, size(t)):
                ny[j] /= dot(ny[j], ny[j]) **.5
                num[j] = dot(ny[j], ddy[j])
                den[j] = dot(dy[j], dy[j])

        k = -num / den
        return k
```

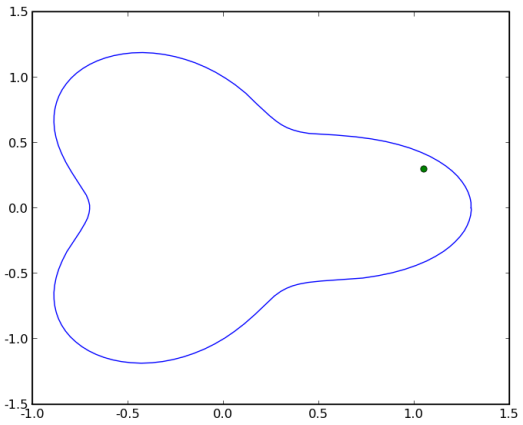
and here is the code for $\frac{\partial \Phi(x,y)}{\partial n_n}$:

```
def fundSolDeriv(x, y, ny):
        if size(shape(x)) < 2: x = x.reshape(1, len(x))
        m = shape(x)[0]

        s = (x - kron(ones((m, 1)),y))
        n = kron(ones((m, 1)), ny)

        r = sqrt(multiply(s,s).sum(axis=1))
        costh = multiply(n,s).sum(axis=1) / r
        nd = (1/(2*pi))*costh / r;
        nd = nd.reshape(m, 1)
        return nd
```

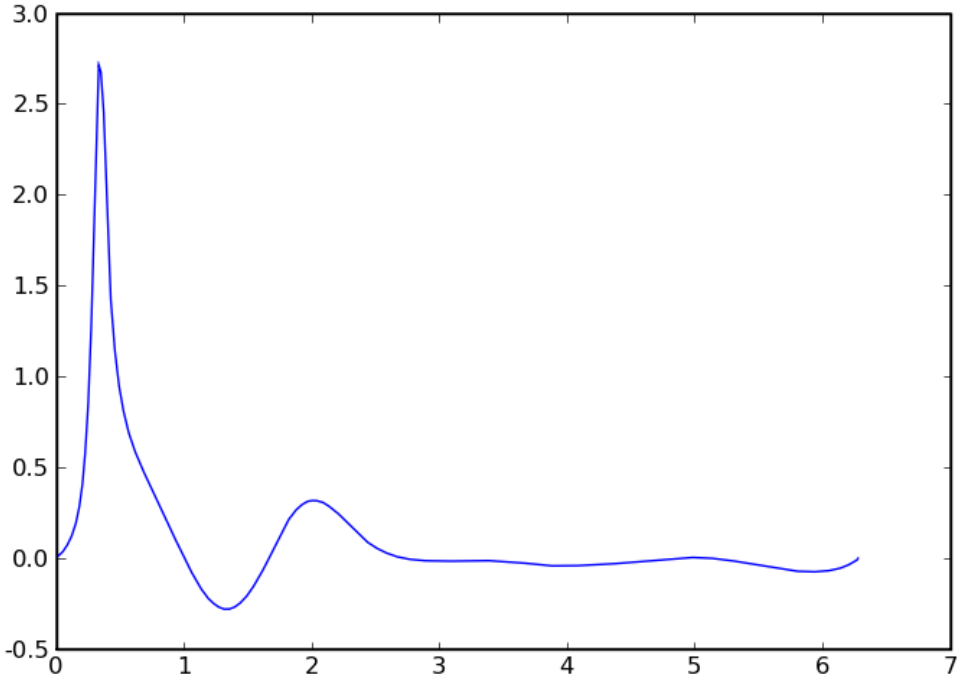
Once we have solved for τ at the nodes, we can integrate over our domain Ω using periodic quadrature. Since we known an exact solution for our BVP we can compute the absolute error between our solution and the true solution. Below is a plot of the log of the absolute error of our solution. We have plotted over a grid with x and y ranging from -1.3 to 1.3 with a step size of 0.05 and N = 50 quadrature points.



Notice the obvious problem with our solution. We have excellent accuracy in the center of Ω, but as we move towards the boundary, the error rapidly approach O(1). This problem arises in the computation of $\frac{\partial\Phi(x,y)}{\partial n_n}$. Computing $\frac{\partial\Phi(x,y)}{\partial n_n}$ requires dividing by $|x-y|$ and, therefore, as x approaches the boundary, $\frac{\partial\Phi(x,y)}{\partial n_n}$ blows up. Consider the point near the boundary below:



Evaluating our solution at this point requires evaluating the integral of the following curve. This curve represents $\frac{\partial\Phi(x,y)}{\partial n_n}$ with $x = (1.05, 0.3)$ evaluated at the nodes along the boundary.



There is a sharp spike in the curve as the boundary nodes move close to x and then the curve moderates again as the nodes move away from x. Hence, most of the area under the curve is concentrated below the spike. This is the source of the error near the boundary. The choice of a quadrature sheme with equally spaced nodes is inadequate to accurately evaluate integrals with a sharp spikes as the one above. Accurate evaluation of our BVP near the boundary will require adaptive quadrature methods to better handle erratic behavior of the boundary integral.

2 Adaptive Quadrature

Adaptive quadrature is a method that can be used with any quadrature scheme to refine the integration of a function to arbitrarily precision. This is achieved by adaptively breaking up the interval of integration into subintervals near rapidly changing sections of the function. The algorithm for adaptive quadrature is quite simple. We first integrate the function over the entire interval using the desired quadrature scheme. We then subdivide the interval into two equal parts and reevaluate the integral over these subdomains. If the original intergal is within some tolerance of the sum of the integrals of the subintervals, then we return. Otherwise we recursively perform adaptive quadrature on each subinterval. Below is the code for adaptive quadrature.

```
def adaptiveQuadrature(func, params, a, b, tol,
   quadRule):
        N = 8
        X, W = quadRule(N, a, b)
        Q = dot(func(X, params), W)

        if depth >= 500: return Q

        m = (b + a) / 2.
        if m < a or m > b: return Q

        X, W = quadRule(N, a, m)
        Q1 = dot(func(X, params), W)
        X, W = quadRule(N, m, b)
        Q2 = dot(func(X, params), W)

        if abs(Q - (Q1 + Q2)) < tol: return Q1 + Q2
```

```
        Q = adaptiveQuadrature(func, params, a, m, tol,
            quadRule) + adaptiveQuadrature(func, params
           , m, b, tol, quadRule)

        return Q
```

```
def gaussianQuad(N, a, b):
        beta = .5/numpy.sqrt(1.-(2.*(numpy.arange(1, N)
           ))**(-2.))
        T = numpy.diag(beta,1) + numpy.diag(beta,-1);
        D, V = scipy.linalg.eig(T);
        x = numpy.real(D)
        i = x.argsort(axis=0)
        x.sort(axis=0)
        w = 2.*V[0].take(i)**2.

        x = (x + 1.) / 2.  * (b - a) + a
        w = w * (b - a) / 2.

        return x, w
```

In the implementation above I am using a gaussian quadrature scheme with 8 nodes on each subinterval. Also notice that I have included a maximum recursion depth. This prevents the program from terminating mid-computation due to exceeding system stack limits.

The effect of performing adaptive quadrature is that we can 'clump' quadrature points near problematic regions of the function and leave only a sparse number of quadrature points along the rest of domain. This allows us to integrate a function up to any desired precision while keeping the number of quadrature points small.

Using adaptive quadrature on our BVP, however, requires one additional step. The adaptive quadrature routine must be able to select nodes at any point along $\partial\Omega$. Currently we only have values for τ at the original 50 nodes we chose. We can make τ into a continuous function by solving an equation from the derivation of the Nystrom method.

$$\tau(x) - 2\sum_{j=1}^{N} k(x, y_j)\tau_j W_j = -2f(x)$$

Which gives us:

$$\tau(x) = 2\sum_{j=1}^{N} k(x, y_j)\tau_j W_j - 2f(x)$$

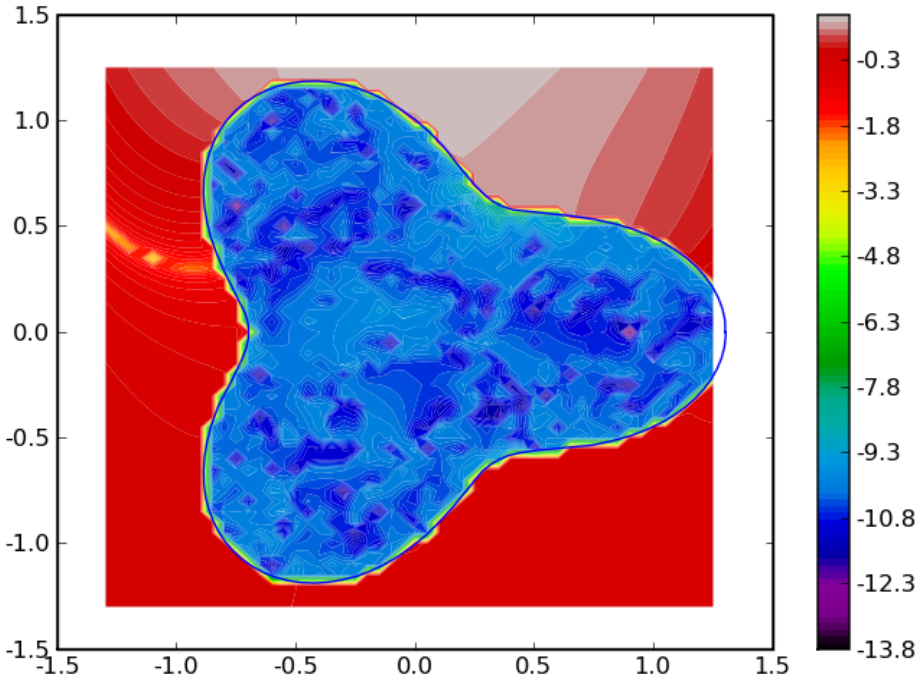
Here is the code for computing the density τ:

```
def density(x, info):
        s = 0
        for j in range(0, len(info.y)):
                speed = sqrt(dot(info.dy[j], info.dy[j
                   ]))
                s += fundSolDeriv(x, info.y[j], info.ny
                   [j]) * info.weights[j] * speed *
                   info.tau[j]

        z = -2. * info.f + 2. * s.ravel()
        return z
```

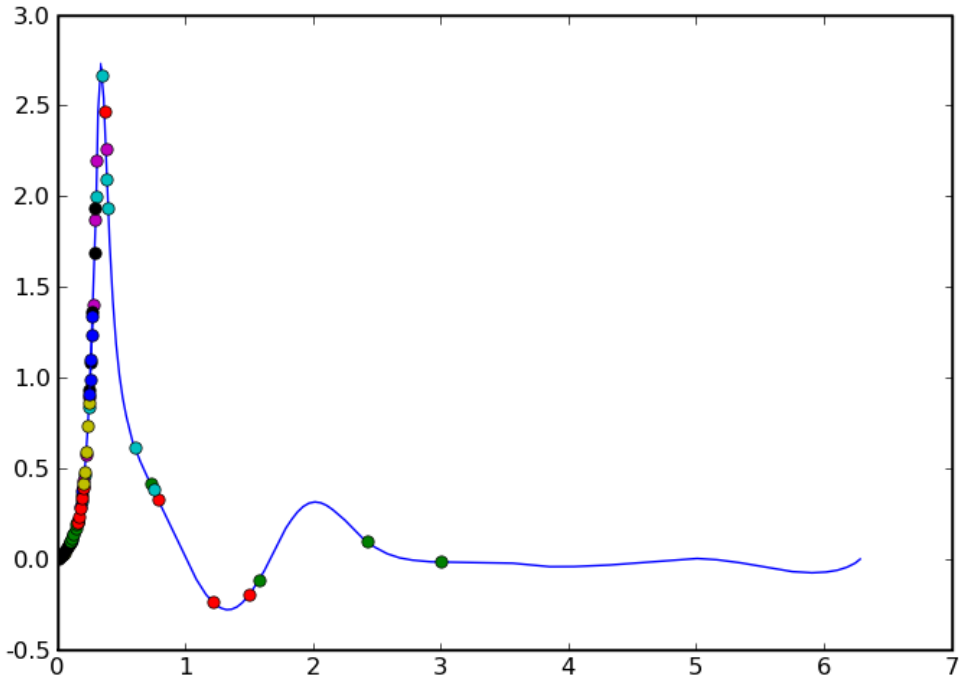
Here is the code for the function we are passing to adaptiveQuadrature for integration:

```
def ker(t, kernelInfo):
        info = kernelInfo.densityInfo
        info.f = f(t)
        x = kernelInfo.x

        y = row_stack([R(t)*cos(t), R(t)*sin(t)]).T
        ny = normal(t)
        den = density(y, info)
        sp = speed(t)

        N = len(t)

        z = zeros(N)
        for j in range(0, N):
                z[j] = fundSolDeriv(x, y[j], ny[j]) *
                   den[j] * sp[j]

        return z
```

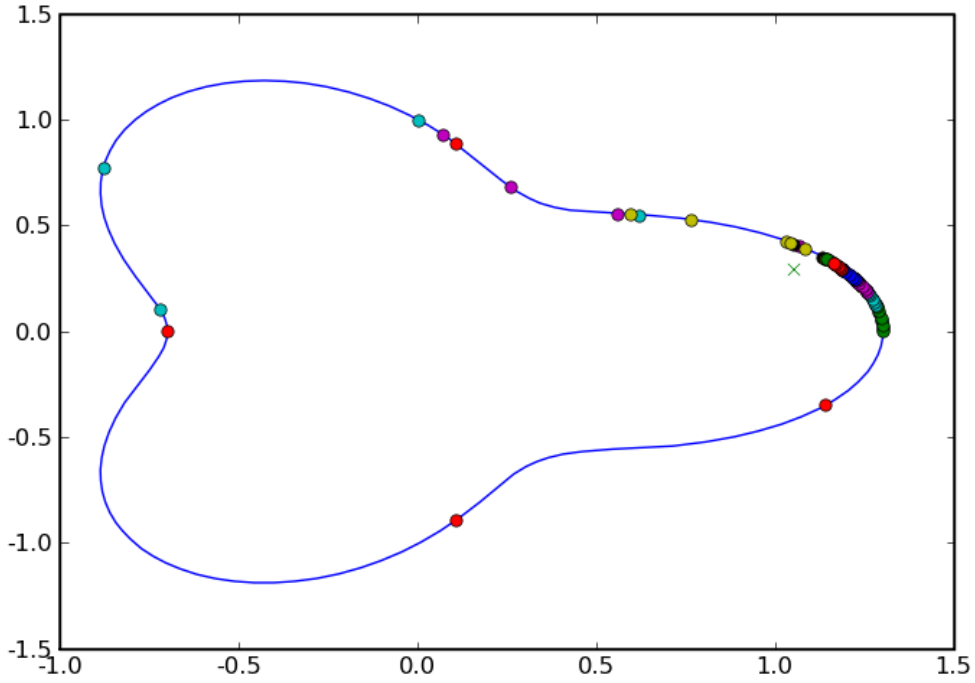
Now we can again evaluate the BVP over Ω, this time, using adaptive quadrature.



Here I have used a tolerance of 10^{-10}. We can see this reflected in the plot as the error in our solution is less than 10^{-10} over the entire interior of Ω. Below is a plot of the boundary integral with the spike from the previous section evaluated to an accuracy of 10 digits at a recursive depth of 8. We can see how the adaptive quadrature method has distributed most of the nodes near the spike.

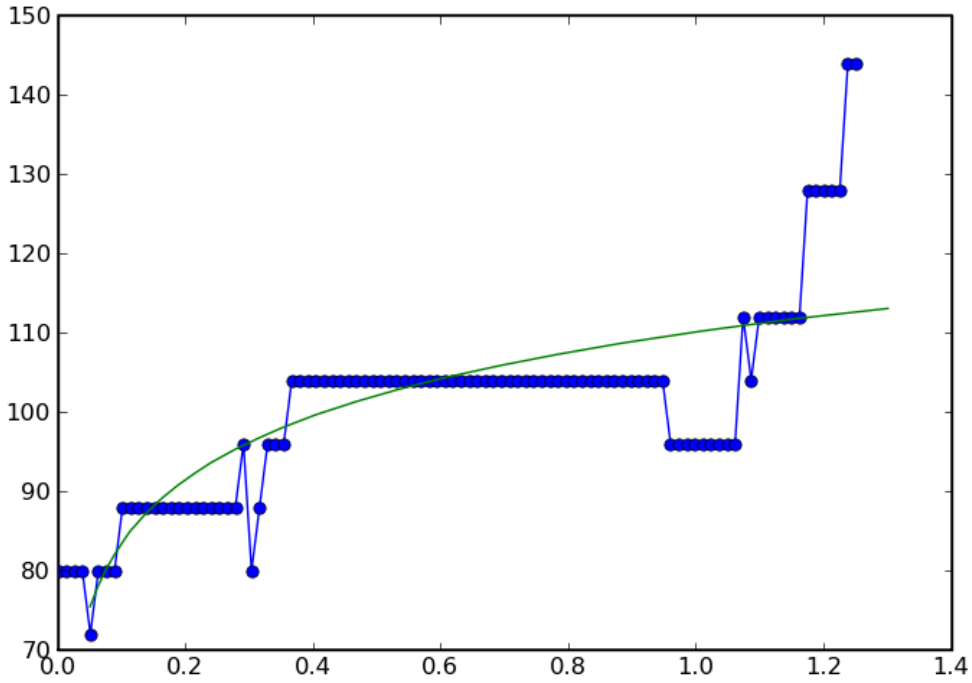


And here are the nodes drawn along the boundary of Ω with a cross marking the position of the point x we are evaluating at.



Despite the significant gains adaptive quadrature has provided us in accuracy near the boundary, there is a significant price to pay in computational time. Our initial solution to the Laplace BVP without adaptive quadrature runs in approximately 1 second. By comparison, the solution with adaptive quadrature requires about 1 hour and 15 minutes to compute. This is with a rather modest grid of 52 by 52 points. A significant factor in this increased computational time is the inability to vectorized the adaptive algorithm. In the non-adaptive case we can vectorize our integration by passing an entire column of our grid to the integration routine to be processed using fast vector algebra routines. With adaptive quadrature, however, the adaptive algorithm chooses nodes based on the point x that is currently being evaluated. For this reason, we can only pass one grid point to be integrated at a time.

Below is a plot that shows the number of quadrature points required to achieve 10 digits of accuracy vs the distance along the x-axis as we approach the boundary. This plot is achieved by sampling 100 points along the x-axis in the interval $[0.0, 1.3)$. The smooth curve is a plot of $8log_2(N)$, where N is is the number of nodes.



This illustrates that the computational effort grows logarithmically as we approach the boundary. This fit holds until we get very close to the boundary at which point we see a sudden jump. It is possible that we see this jump very close to the boundary because the spike in the integrand is so severe that our subintervals approach machine precision without satisfying the required tolerance. Here we have chosen to fit our data with $8log_2(N)$ because we split our intervals in two at each step giving us base 2 and we split each subinterval up into 8 nodes. Also notice that this data isn't distance to the boundary vs nodes as we would like. Instead we have distance along the x-axis which doesn't approach the boundary uniformely do to the wobble in our domain. Hence, this data likely has more noise than a plot that represented true distance to the boundary.

Notice that the number of required nodes starts at around 80 at the point (0, 0). The non-adaptive plot has an accuracy of 11 digits in this region with only the original 50 nodes. This indicates that for points sufficiently far away from the boundary, we do not need to perform adaptive quadrature. If we consider points along the x-axis, we don't need to perform adaptive quadrature on the interval $[0, 0.5]$. If we write a routine to determine the true distance to there boundary, we may be able to skip adaptive quadrature on a significant portion of our grid giving us a significant speedup in the overall computation.