

Summary of algorithm

August 26, 2022

Let $Z_1, Z_2, Z_3 (= Z_{j,k}, Z_{j+1,k}, \dots)$ be any of the triangles in the logarithmic Z -plane. Suppose that we have mapped the whole mesh approximately ν -conformally, and these three vertices go to W_1, W_2, W_3 . We want to find a better approximation W_1^*, W_2^*, W_3^* .

1. Preliminary calculations.

Given a constant Beltrami coefficient ν for this triangle, there is a ν -conformal mapping to the triangle $0, 1, p$ where $p = (Z_3 - Z_1)/(Z_3 - Z_2)$. The map is

$$Z \mapsto V = \frac{1}{1 + \nu} \left(\frac{Z - Z_1}{Z_2 - Z_1} + \nu \overline{\left(\frac{Z - Z_1}{Z_2 - Z_1} \right)} \right).$$

Thus the value of V at Z_3 is

$$p = \frac{1}{1 + \nu} \left(\frac{Z_1 - Z_3}{Z_1 - Z_2} + \nu \overline{\frac{Z_1 - Z_3}{Z_1 - Z_2}} \right). \quad (1)$$

The first step is to determine a conformal linear mapping

$$W = aV + b$$

which takes $0, 1, p$ to W'_1, W'_2, W'_3 , which can be considered as functions of (a, b) . The definition of $a = a_0 + ia_1$, $b = b_0 + ib_1$ is to minimize the discrepancy

$$\Phi(a, b) = |W'_1 - W_1|^2 + |W'_2 - W_2|^2 + |W'_3 - W_3|^2$$

(Thus if the map $Z_1, Z_2, Z_3 \mapsto W_1, W_2, W_3$ was already ν -conformal, the minimum of Φ will be zero, because $a = 1$, $b = 0$, $W'_i = W_i$.)

It is very simple to write Φ explicitly as a quadratic polynomial in a_0, a_1, b_0, b_1 with coefficients in terms of $p, u_1, u_2, u_3, v_1, v_2, v_3$ where $W_1 = u_1 + iv_1$, etc. The formula is

$$\begin{aligned} \Phi(a, b) = & (|w_1| + |w_2| + |w_3|)^2 \\ & - 2a_0(p_0u_3 + p_1v_3 + u_2) + 2a_1(p_1u_3 - p_0v_3 - v_2) \\ & - 2b_0 \operatorname{Re}(w_1 + w_2 + w_3) - 2b_1 \operatorname{Im}(w_1 + w_2 + w_3) \\ & + |a|^2(|p|^2 + 1) + 3|b|^2 \\ & + 2(a_0b_0 + a_1b_1)(1 + p_0) + 2p_1(a_0b_1 - a_1b_0). \end{aligned}$$

Then it is simple to write down the gradient $\nabla\Phi$, and the minimum can be found by solving for $\nabla\Phi = 0$ with an explicit formula. With respect to the variables a_0, a_1, b_0, b_1 ,

$$\begin{aligned}\nabla\Phi = & -(u_2 + p_0 u_3 + p_1 v_3) + 2(1 + p_0^2 + p_1^2)a_0 + 2(1 + p_0)b_0 + 2p_1 b_1, \\ & -(v_2 + p_0 v_3 - p_1 u_3) + 2(1 + p_0^2 + p_1^2)a_1 - 2p_1 b_0 + 2(1 + p_0)b_1, \\ & -2(u_1 + u_2 + u_3) + 2(1 + p_0)a_0 - 2p_1 a_1 + 6b_0, \\ & -2(v_1 + v_2 + v_3) + 2p_1 a_0 + 2(1 + p_0)a_1 + 6b_1\end{aligned}\tag{2}$$

which is linear. Thus $\nabla\Phi(a_0, a_1, b_0, b_1) = 0$ when

$$\begin{aligned}a_0 &= \frac{-(u_1 - 2u_2 + u_3) - p_0(u_1 + u_2 - 2u_3) - p_1(v_1 + v_2 - 2v_3)}{\delta}, \\ a_1 &= \frac{-(v_1 - 2v_2 + v_3) - p_0(v_1 + v_2 - 2v_3) + p_1(u_1 + u_2 - 2u_3)}{\delta}, \\ b_0 &= \frac{(u_1 + u_3) - p_0(u_2 + u_3) + p_1(v_2 - v_3) + (p_0^2 + p_1^2)(u_1 + u_2)}{\delta}, \\ b_1 &= \frac{(v_1 + v_3) - p_0(v_2 + v_3) - p_1(u_2 - u_3) + (p_0^2 + p_1^2)(v_1 + v_2)}{2(p_0^2 + p_1^2 - p_0 + 1)}\end{aligned}\tag{3}$$

where

$$\delta = 2(p_0^2 + p_1^2 - p_0 + 1).\tag{4}$$

Applying these optimum values of a, b , the the optimal values of W'_1, W'_2, W'_3 as the V -images of $0, 1, p$ are

$$\begin{aligned}W_1^* &= b, \\ W_2^* &= a + b, \\ W_3^* &= ap + b.\end{aligned}\tag{5}$$

Lemma. If the indices 1,2,3 are permuted, the same optimal values will be obtained (after applying the same permutation).

2. Phase 1.

The *first phase* of the algorithm is to apply this process to each triangle τ_{jk}^\pm of the W -mesh (notation of our article), giving a new collection of triangles σ_{jk}^\pm . We can say that σ_{jk}^\pm is the “ ν -adjustment” of τ_{jk}^\pm .

[Eliminate the previously mentioned step of averaging the values of a vertex according the the six neighboring triangles.] We will use $\{\sigma_{jk}^\pm\}$ to construct a new mesh W_{jk}^* .

3. Reasoning for the following construction..

Consider that the original τ_{jk}^\pm have the following properties:

- they fit together perfectly, but
- they do not have exactly the correct ν -shapes

This is because they come from a mesh which is only an approximate solution to the Beltrami equation.

On the other hand, the σ_{jk}^\pm satisfy

- they have more nearly correct ν -shapes, but
- they do not fit together perfectly,

This is because of the optimization of Φ which is a local property; it would require global information for them to fit together. To understand this point more clearly (if that is necessary), think of a constant $\nu > 0$ (such as $\nu = 0.3$) applied to the original Z -triangulation. All of the equilateral triangles are mapped to isosceles triangles with a vertical edge opposite to the (left or right) apex. These triangles are wider than the equilateral triangles and have less height. They have approximately the same center as the original ones, so they overlap each other to the left and right, and leave empty spaces above and below. It is obvious that by moving them properly they will all fit together into a large rectangular array which is wider than it is high.

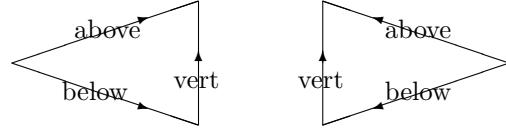
The idea of “fitting together” the σ_{jk}^\pm means applying conformal mappings $\alpha\sigma_{jk}^\pm + \beta$ (this means $\alpha = \beta_{jk}^\pm$ and $\beta = \beta_{jk}^\pm$ for simplicity). The size/rotation data $\alpha \in \mathbb{C} \setminus \{0\}$ depends only on the immediate neighbors of a triangle, because that is what it has to fit with. Since τ_{jk}^\pm fits its neighbors perfectly, and σ_{jk}^\pm is deduced from those neighbors, it seems reasonable that σ_{jk}^\pm is already approximately of the same size and rotation. Thus we assume $\alpha = 1$ for all practical purposes. The essential problem is to determine β .

4. Phase 2.

From this observation the *second phase* of the algorithm is very natural. Basically we just line up the triangles by increasing values of $-j$. Thus we must think in terms of displacements.

The “side differences” or “side displacements” or “side vectors” of the triangles are defined by

$$\begin{aligned}\sigma_{jk}^{\pm}[\text{vert}] &= \text{highest point} - \text{lowest point}, \\ \sigma_{jl}^{\pm}[\text{above}] &= \text{highest point} - \text{apex}, \\ \sigma_{jl}^{\pm}[\text{below}] &= \text{lowest point} - \text{apex}.\end{aligned}$$



The apex of a rightward (leftward) pointing triangle is its rightmost (leftmost) point. The apex of τ_{jk}^{\pm} is Z_{jk} . It is straightforward to write down the formulas in terms of the W_{jk} coordinates of “ W_1, W_2, W_3 ” described in the the first phase:

For j even,

$$\begin{aligned}\sigma_{jk}^+[\text{vert}] &= W_{j-1,k}^{\sigma} - W_{j-1,k-1}^{\sigma}, \\ \sigma_{jk}^-[\text{vert}] &= W_{j+1,k}^{\sigma} - W_{j+1,k-1}^{\sigma}, \\ \sigma_{jk}^+[\text{above}] &= W_{j-1,k}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^-[\text{above}] &= W_{j+1,k}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^+[\text{below}] &= W_{j-1,k-1}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^-[\text{below}] &= W_{j+1,k-1}^{\sigma} - W_{j,k}^{\sigma}.\end{aligned}\tag{6}$$

For j odd,

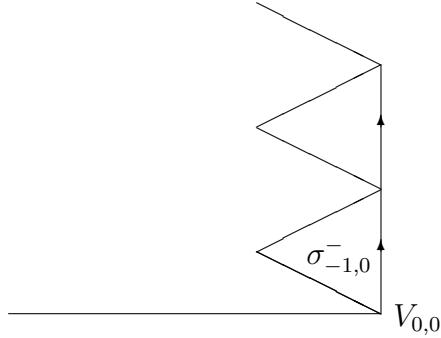
$$\begin{aligned}\sigma_{jk}^+[\text{vert}] &= W_{j-1,k+1}^{\sigma} - W_{j-1,k}^{\sigma}, \\ \sigma_{jk}^-[\text{vert}] &= W_{j+1,k+1}^{\sigma} - W_{j+1,k}^{\sigma}, \\ \sigma_{jk}^+[\text{above}] &= W_{j-1,k+1}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^-[\text{above}] &= W_{j+1,k+1}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^+[\text{below}] &= W_{j-1,k}^{\sigma} - W_{j,k}^{\sigma}, \\ \sigma_{jk}^-[\text{below}] &= W_{j+1,k+1}^{\sigma} - W_{j,k}^{\sigma}.\end{aligned}\tag{7}$$

In this notation, the symbol W^σ is just a reminder that the corresponding value must be taken from the calculated triangle σ_{jk} , since there are actually 6 triangles in general sharing a single vertex.

To determine the vertices in the formula, remember that in *Mathematica*, `graphdomain` plots some vertices and some triangles which do not exist in the triangulation (using $2\pi i$ -periodicity, which `logdomextract` does automatically), and which cause great confusion. There are no triangles $\tau_{0,k}^-$ or $\tau_{-N,k}^+$. The topmost $\tau_{j,N-1}$ (j odd) and the bottommost $\tau_{j,0}$ contain vertices which are not in the triangulation. Although the [above] of one triangle is (-1) times the [below] of another, for the approximately calculated $\sigma_{j,k}^\pm$) these are not exactly the same.

Step 1. First we set up the values on the imaginary axis. This is a recursive definition:

$$\begin{aligned} V_{00} &= 0, \\ V_{0k} &= i \operatorname{Im}(V_{0,k-1} + \sigma_{-1,k}^-[\text{vert}]), \quad k = 1, \dots, N-1. \end{aligned} \tag{8}$$



This just says to stack up the vertical edges of the leftward pointing triangles at the right side of the mesh. Note that we are throwing away the real parts, which are small.

We need one more value, which is essentially “ V_{0N} ” but not properly belonging to the mesh,

$$V^{\text{top}} = i \operatorname{Im}(V_{0,N-1} + \sigma_{-1,N-1}^-[\text{vert}]),$$

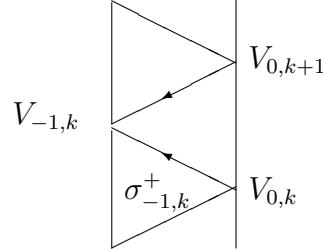
While we do not worry yet about whether the total height of the N edges is 2π , we need to save this imaginary height:

$$dV = V^{\text{top}} - V_{0,0} = V^{\text{top}}.$$

Subsequent steps. Next, for $j = -1$, we want

$$V_{-1,k} = \frac{1}{2}((V_{0,k} + \sigma_{-1,k}^+[\text{above}]) + (V_{0,k+1} + \sigma_{-1,k+1}^+[\text{below}])), \quad k = 0, \dots, N-2,$$

$$V_{-1,N-1} = \frac{1}{2}((V_{0,N-1} + \sigma_{-1,0}^+[\text{above}]) + (V^{\text{top}} + \sigma_{-1,0}^+[\text{below}])).$$



In other words, we use the images of the two rightward pointing triangles which “have a vertex at $Z_{-1,k}$ ” and anchor the displacements at the $j = 0$ points which are already established. Then we average their values to find $V_{-1,k}$. Unlike $j = 0$, now the order in which we calculate the new points for $j = 1$ does not matter. (The column is not “recursive”.)

We also need to save the extra values

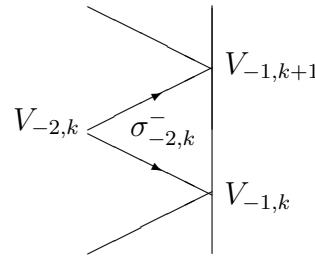
$$V^{\text{bot}} = \frac{1}{2}(((V_{0,N-1} - dV) + \sigma_{-1,N-1}^+[\text{above}]) + (V_{0,0} + \sigma_{-1,0}^+[\text{below}])),$$

$$dV = V_{-1,N-1} - V^{\text{bot}}.$$

Then for $j = -2$, we use leftward pointing triangles in an analogous way:

$$V_{-2,0} = \frac{1}{2}((V_{-1,0} - \sigma_{-2,0}^-[\text{below}]) + (V^{\text{bot}} - \sigma_{-2,N-1}^-[\text{above}])),$$

$$V_{-2,k} = \frac{1}{2}((V_{-1,k-1} - \sigma_{-2,k}^-[\text{below}]) + (V_{-1,k} - \sigma_{-2,k}^-[\text{above}])), \quad k = 0, \dots, N-1,$$



This time, each new vertex $V_{-2,k}$ uses displacements from only one triangle, of which it is the apex. Note that we subtract the displacements because they emanate from the apex. Also, for the next column,

$$V^{\text{top}} = \frac{1}{2}(((V_{-1,N-1} - \sigma_{-1,0}^-[\text{below}]) + (V_{-1,0} + dV) + \sigma_{-1,0}^-[\text{above}])),$$

$$dV = V^{\text{top}} - V_{-2,0}.$$

In general, for $j < 0$ odd,

$$V_{j,k} = \frac{1}{2}((V_{j+1,k} + \sigma_{j,k}^+[\text{above}]) + (V_{j+1,k+1} + \sigma_{j,k+1}^+[\text{below}])), \quad k = 0 \dots, N-2,$$

$$V_{j,N-1} = \frac{1}{2}((V_{j+1,N-1} + \sigma_{j,0}^+[\text{above}]) + (V^{\text{top}} + \sigma_{j,0}^+[\text{below}])),$$

$$V^{\text{bot}} = \frac{1}{2}((V_{j+1,N-1} - dV) + (\sigma_{j,N-1}^+[\text{above}]) + (V_{j+1,0} + \sigma_{j,0}^+[\text{below}])),$$

$$dV = V_{j,N-1} - V^{\text{bot}}. \quad (9)$$

and for j even,

$$V_{j,0} = \frac{1}{2}((V_{j+1,0} - \sigma_{j,0}^-[\text{below}]) + (V^{\text{bot}} - \sigma_{j,N-1}^-[\text{above}])),$$

$$V_{j,k} = \frac{1}{2}((V_{j+1,k-1} - \sigma_{j,k}^-[\text{below}]) + (V_{j+1,k} - \sigma_{j,k}^-[\text{above}])) \quad k = 1, \dots, N-1,$$

$$V^{\text{top}} = \frac{1}{2}((V_{j+1,N-1} - \sigma_{j,0}^-[\text{below}]) + (V_{j+1,0} + dV) - \sigma_{j,0}^-[\text{above}]),$$

$$dV = V^{\text{top}} - V_{j,0}. \quad (10)$$

If there are any mistakes in the indices or signs, then one should follow the picture and not the indices written here!

5. Alternatives.

There were some arbitrary choices made in this construction, and it could be done differently. The construction of V_{0k} only uses leftward pointing triangles. Perhaps a more accurate construction would be obtained using right-pointed triangles also. This could be done by adjusting the values of V_{0k} after constructing the $V_{-1,k}$. It is important for the V_{0k} to be accurate because the rest of the mesh depends on them.

Similarly each $V_{-1,j}$ was constructed using two rightward pointing triangles. One could also use the leftward pointing triangle which has $W_{-1,j}$ at its apex (i.e. to take the average of three points instead of two). I don't think this would make much difference for a fine mesh, because the neighboring triangles are constructed to fit together very well. (If the algorithm works, one of the questions in the proof will be to study how well they really fit together.)

6. Phase 3.

The values still have to be normalized vertically, to make them cyclic of period $2\pi i$. We can call this *phase three*, the construction of W_{jk}^* .

For each j , let $C_j = (V_{jN} - V_{j0})/2$. Write down the affine-linear mapping $\alpha_j W + \beta_j$ which takes the segment $[V_{j0}, V_{jN}]$ to the vertical segment $[C_j - \pi i, C_j + \pi i]$. Replace all of the β_j with $\beta_j + c$ where c is chosen so that $\alpha_0 V_{00} + \beta_0 = 0$. Then apply the j -th mapping to every V_{jk} ($0 \leq k \leq N$) to get

$$W_{jk}^* = \alpha_j V_{jk} + \beta_j. \quad (11)$$

It should be clear that if the mesh is very fine, the mappings are close to each other when the values of j are close. The individual mappings (11) are all conformal, and by the “closeness”, the third phase is approximately a conformal operation.

The final mesh $\{W_{jk}^*\}$ is now $2\pi i$ -periodic (so we can throw away the values W_{jN}^*). It is also normalized by $W_{00}^* = 0$ and $\operatorname{Re} W_{0k} = 0$. Of course, what we would like is for the mesh to be 1-to-1, that is, all of the image triangles are disjoint.

Phases 1,2,3 make one iteration. The calculation time of an iteration is $O(MN)$ for an $M \times N$ grid because the procedure is essentially local (this means the calculation of each σ_{jk}^* is $O(1)$), and because similar considerations apply to phases 2 and 3. The calculation time to get to a good solution is very different from the solution of a large linear system because now the number of iterations will depend on how big or how complicated the Beltrami coefficient is.

I think that if ν is constant (at least if it is real), then we will arrive at the correct answer in one iteration, because of the symmetry of the triangular mesh. I don't know how fast it will converge in general, but I hope it will be approximately linear until it gets close to the best possible approximation for a given mesh (then it will not get any closer).

Another way to use the idea is to start with small values of M, N (coarse mesh), apply one or a few iterations, and then double M, N and again apply one or a few iterations. If

the idea above for a fixed mesh is correct, then this version of the algorithm will converge to the true solution since an arbitrarily fine mesh allows an arbitrarily good solution.

Another technique is to apply the algorithm to small increments such as $0.1\nu, 0.2\nu, \dots$, using the result of each phase for initial guess of the next phase. (Homotopy method.)

PROGRAMMING SCHEME

Write the function `newtriangle(nu,z1,z2,z3,w1,w2,w3)` as described in “phase one” of the iteration above. This means calculate p via (1) and then a_0, a_1, b_0, b_1 via (3) and (4), where u_i, v_i are parts of w_1, w_2, w_3 , and then apply (5). Construct the list of triangles σ_{jk}^\pm indexed by (j, k, \pm) . Write functions `vert`, `above`, `below` which give the edge displacements of a triangle. Then construct the mesh V_{jk} of phase two and normalize as in phase three.

```

iteration()
  # phase 1
  For j,k running through whole mesh and pm = -1,1:
    sigma(j,k,pm) <-- newtriangle(nu,...)
  # phase 2
  v[0,0] <-- 0
  For k running through 1,...,N
    v[0,k] <-- v[0,k-1] + vert( sigma[...] )
  For j running through -1,...,-M
    For k running through 0,...,N
      If(j is even) v[j,k] <-- (1/2)( .... ) ...
      else           v[j,k] <-   ...
  # phase 3
  For j running through 0,...,-M
    alpha <-- ...
    beta  <-- ..
    If(j=0) c = ... # normalization
    beta  <-- beta + c
  For j running through -1,...,-M # the order doesn't matter
    For k running through 0,...,N-1
      wnew[j,k] = alpha*vnew[j,k] + beta

```

and perhaps a loop at the end with `w[j,k]=wnew[j,k]` to prepare for the next iteration.