# Project Zipper Final Report Spring 2006

Petr Moravsky under supervision of Tom Kennedy

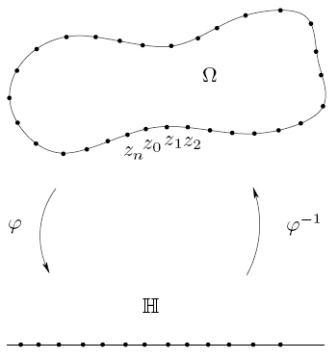
May 31, 2006

## Introduction

One of the most important problems in complex analysis is to find a conformal map of a complicated region to simpler closed region so that certain computations can be done in the simpler region, and then the results can be applied to a complicated region. The Zipper algorithm helps us to solve this problem in 2D space.

## Algorithm Overview

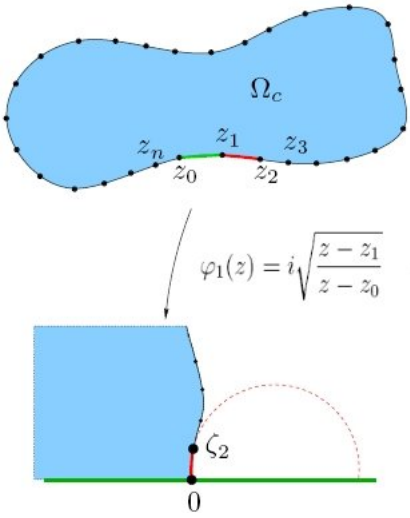
The Zipper algorithm computes numerically a conformal map of complex region, given by a set of points on the boundary, to a unit disk. Then if we want, we can map the disk to half plane. Of course, if we find such maps for 2 regions, then it is not hard to take the inverse of one of the maps and map one region to another.



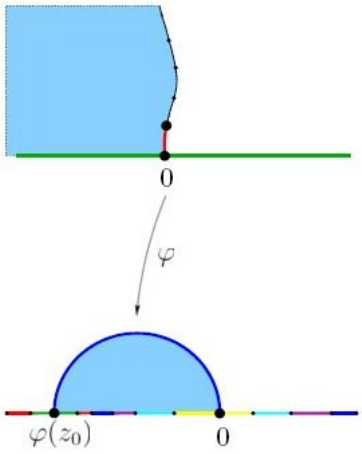
First we apply $\varphi_1$ to each point z to map the region to an open subset of the half plane.

$$\varphi_1(z) = i\sqrt{\frac{z - z_1}{z - z_0}}$$

The point $z_0$ goes to $\infty$, and the point $z_1$ goes to 0. This is called the unzipping step.

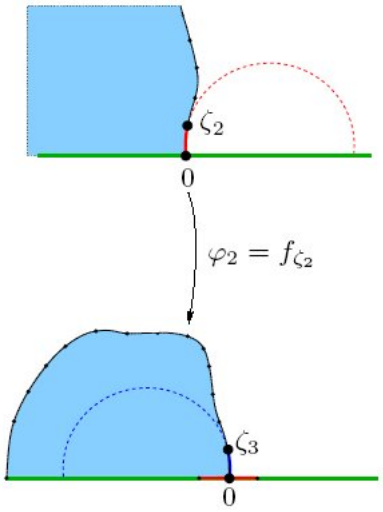



Now we want to map the open region to the half disk. In order to do so, we need to find $f : \mathbb{H}\backslash\gamma \to \mathbb{D}$ that sends most of the boundary to the real axis and maps the last piece $z_0 z_n$ to the half circle .

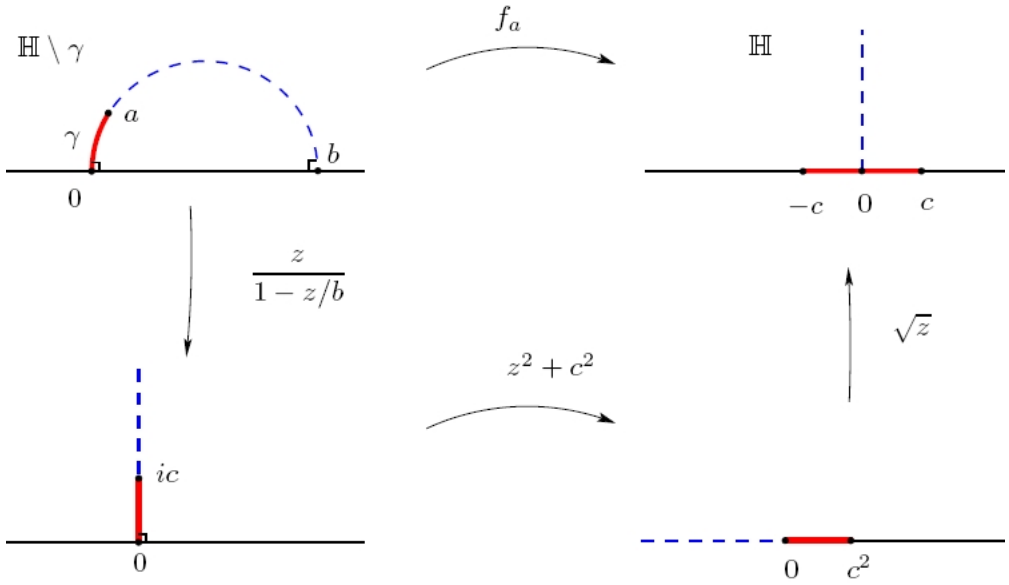


The boundary can be a complicated curve so we have to find our map approximatelly. Since the boundary consists of many curved segments, our job is to apply a composition of maps, each of which removes a single piece of the boundary from the half plane $f_a : \mathbb{H}\backslash\gamma_a \to \mathbb{H}$. Notice an important feature of map $f_a(z) = \sqrt{\left(\frac{z}{1-\frac{z}{b}}\right)^2 + c^2}$, $b = \frac{|a|^2}{\mathrm{Re}a}$, $c = \frac{|a|^2}{\mathrm{Im}a}$: it takes the point at infinity to some finite point on the real axis so after removing the first piece of the slit all points are finite. More specifically,

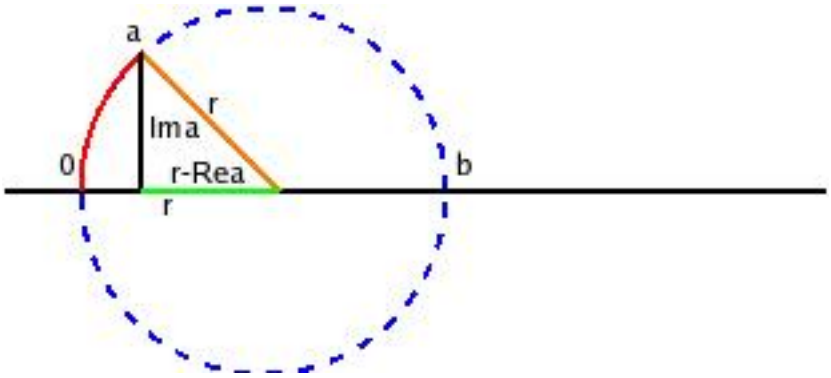
$$f_a(\infty) = \sqrt{\left(\frac{1}{\frac{1}{\infty} - \frac{1}{b}}\right)^2 + c^2} = \sqrt{b^2 + c^2} = \sqrt{\frac{|a|^4}{\mathrm{Re}a^2} + \frac{|a|^4}{\mathrm{Im}a^2}}$$
$$= \frac{|a|^2}{\mathrm{Re}a\mathrm{Im}a}\sqrt{\mathrm{Re}a^2 + \mathrm{Im}a^2} = \frac{|a|^3}{\mathrm{Re}a\mathrm{Im}a}$$




Unfortunatelly, there is no simple map taking a single curved segment to the real axis, while preserving all the points in the upper half plane in the upper half plane. However, after the unzip step $z_0 \dots z_n$ consists of many curved segments, and if they are small, we can think of $z_0 \dots z_n$ consisting of circular arcs where the first arc, $\gamma_a$ is an arc of a circle from 0 to $a \in \mathbb{H}$ which is orthogonal to R at 0 and at $b \in \mathbb{R}$. Also if our curved segments are not small, we can add additional points on the original curve to make sure that they are. Consider the composition of maps shown on a figure bellow:




The first map, $f(z) = \frac{z}{1-\frac{z}{b}}$, is a linear fractional transformation that maps the curved segment to the straight segment. The second map $f(z) = z^2 + c^2$ rotates and shifts the straight segment to make sure it gets mapped to the real axis. The third map $f(z) = \sqrt{z}$ is needed to fold back the plane and make that all the points remain in the upper half plane.



$$|x-r| = r$$
$$|a-r| = r$$
$$|a-r|^2 = r^2$$
$$(\mathrm{Re}a - r)^2 + \mathrm{Im}^2 a = r^2$$
$$\mathrm{Re}^2 a - 2\mathrm{Re}ar + \cancel{r^2} + \mathrm{Im}^2 a = \cancel{r^2}$$
$$r = \frac{\mathrm{Re}^2 a + \mathrm{Im}^2 a}{2\mathrm{Re}a}$$
$$b = 2r = \frac{\mathrm{Re}^2 a + \mathrm{Im}^2 a}{2\mathrm{Re}a} = \frac{|a|^2}{\mathrm{Re}a}$$

$$f(a) = f(x+iy) = \frac{x+iy}{1 - \frac{(x+iy)x}{x^2+y^2}} =$$
$$\frac{(x+iy)(x^2+y^2)}{\cancel{x^2} + y^2 - \cancel{x^2} - ixy} = \frac{(x+iy)(x^2+y^2)}{y(y-ix)} =$$
$$\frac{(x+iy)(x^2+y^2)}{y(y-ix)} \frac{y+ix}{y+ix} =$$
$$\frac{(ix^2 + iy^2 + \cancel{xy} - \cancel{xy})}{\cancel{y(x^2+y^2)}} \cancel{(x^2+y^2)} = i\frac{|z|^2}{\mathrm{Im}z} = ic$$

The running time of the algorithm is determined by the fact that in order to compute $\varphi_2$, we need to compute $\varphi_1(z_2)$. In order to compute $\varphi_3$, we need to compute $\varphi_2\varphi_1(z_3)$. In order to compute $\varphi_4$, we need to compute $\varphi_3\varphi_2\varphi_1(z_3)$, and so on. Therefore, the total time to compute our composition is governed by the recurrence $T(n) = T(n-1) + O(n)$ so $T(n)$ is quadratic.

# Implementation

Currently, the basic algorithm has been implemented as a Java program. During the implementation two major issues arrose. They both have to do with the choice of Java as a programming language.

I chose Java because of the "Compile once, run anywhere" slogan. This way I could have working implementations for UNIX and Windows without changing any of the code. Unfortunately, it turned out the slogan was limited to Sun J2SE Runtime Environment (JRE). The code runs with no problems on Windows with Sun JRE 5.0 installed, but most popular Linux distributions such as Fedora Core 5 include the GNU version of the JRE, which is not compatible with the Sun version. Currently, Sun version of the JRE needs to be installed in order to run the software on Linux. However, the GNU version of the JRE is undergoing rapid development so I expect this issue to become moot in a few months.

The second issue has to do with Java being an object-oriented language and object creaton overhead. Complex numbers are created as an

immutable type, meaning once an instance of Complex object is created, it cannot be modified. If we use code like this for complex numbers:

```
public Complex add(Complex z)
{
    return new Complex(z.Re+Re, z.Im+Im);
}
public Complex sub(Complex z)
{
    return new Complex(Re-z.Re, Im-z.Im);
}
public Complex mul(Complex z)
{
    return new Complex(Re*z.Re-Im*z.Im, Im*z.Re+Re*z.Im);
}
```

And then use it as:

```
public Complex map(Complex z)
{
    Complex t1 = z.sub(z1);
    Complex t2 = z.sub(z0);
    Complex t3 = t1.div(t2);
    Complex t4 = t3.sqrt2Pi();
    Complex t5 = t4.mul(new Complex(0,1));
    return t5;
}
```

We can see that a lot of temprorary objects are being created only to be discarded right after their use. The map() function can be used thousands of times(each time for every point) and object creation overhead can get quite large. This is inefficient, but leads to less bugs. Another way to implement Complex numbers would be to create them as mutable types and define operations like this:

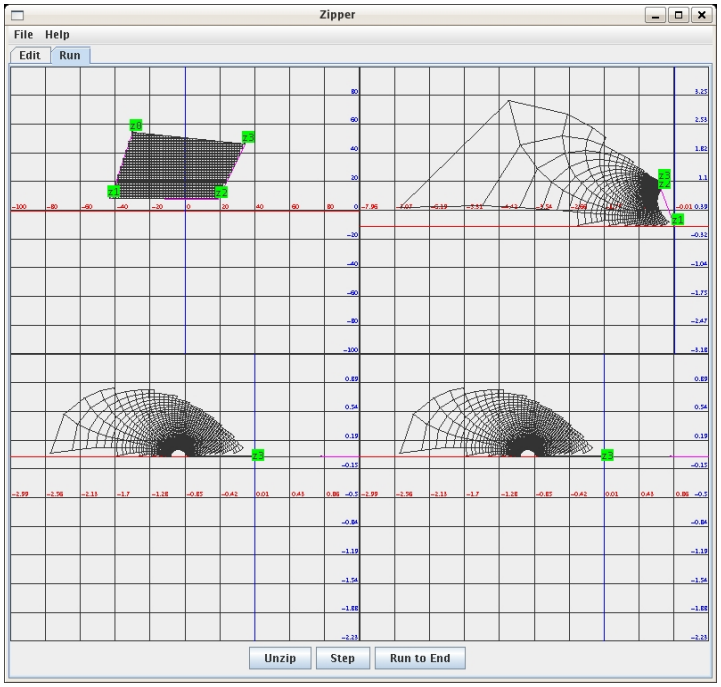
```
public void add(Complex z)
{
    Re = z.Re+Re;
    Im = z.Im+Im;
}
public void sub(Complex z)
```

```
{
    Re = Re-z.Re;
    Im = Im-z.Im;
}
public void mul(Complex z)
{
    Re=Re*z.Re-Im*z.Im;
    Im=Im*z.Re+Re*z.Im;
}
```

But in that case we need to make sure that no undesirable sideffects occur. In our examle if we do a.mul(a) then twhen Im value is calculated, it uses the new Re value, not the old, so the old Re value must be saved and the code becomes more difficult to write. However, if needed switching to mutable types can provide performance improvement.

# Results

Bellow is a screen shot of our program. The image in the top left corner is



the original polygon with the grid. The image in the top right corner is the grid after the unziping step. Two images on the bottom are the final result after the whole composition has been computed.

# Future Work

As it was said earlier, the running time is quadratic. The total running time is determined by the time needed to compute the composition $\varphi(z) = \varphi_n \circ \varphi_{n-1} \circ \ldots \circ \varphi_2 \circ \varphi_1(z)$ where each stage takes $O(k)$ where k is the stage number. In order to improve that running time we plan to use Laurent series representation of our function. Assume that $f_a(z) = \sum_\alpha c_\alpha z^\alpha$ and $f_b(z) = \sum_\beta c_\beta z^\beta$. Then $f_b \circ f_a(z) = \sum_\beta c_\beta \left( \sum_\alpha c_\alpha z^\alpha \right)^\beta$. In that case each stage only takes O(1) plus some time to do series manipulations to compute the composition. Naturally, we plan to keep only a reasonable number of high order terms in the series at each stage (we're planning 12). We believe that this approach will produce significant improvements in running time.

Also as you can see on the provided screen shot. The final result is not a perfect half disk(but then there is some space between the original polygon and the grid). We plan to try some different maps for unziping and transforming to achieve better convergence.

# References

"Convergence of the Zipper algorithm for conformal mapping"
Donald E. Marshall and Steffen Rohde
Department of Mathematics University of Washington