Extending Riemann Mapping Capabilities for the Sage Mathematics Package

Ethan Van Andel Calvin College 2011

Overview

This senior honor project focused on expanding and improving the riemann module that I developed as part of research project at Calvin College in the summer of 2009. riemann is part of the Sage software system and provides computation and visualization tools for Riemann mapping—an important component of complex analysis. The riemann module is the only publicly available implementation of Riemann mapping. Its primary use is educational and explorational, although it could be applied to scientific research as well.

This project involved a great deal of mathematical and computer science research to develop and implement the techniques needed. I have attempted to provide a thorough explanation of the relevant mathematical background. Nevertheless, readers without a background in complex analysis may be lost at times. Generally, if the reader finds terminology that they want defined more thorougly, Wikipedia is an effective resource for filling that gap.

In addition, since this project is highly interwoven with the 2009 research, some of the work described in this report was actually done two years ago. I have tried to keep the focus on the work for this project, discussing previous work only when it is relevant to an understanding of the whole. To avoid unnecessary complications, strict distinctions will not be made as to the exact place of every bit of work. As a rough guide, the work that was done for this project includes some general and usability improvements, substantial optimization, the exterior Riemann map, the Ahlfors spiderweb, and analytic error testing.

Background

Sage

Sage is an open source mathematics software system. Since its inception in 2005 Sage has rapidly grown in popularity. Its free, open-source nature, and the ability to provide services to a large user base over the web makes it appealing to a wide variety of users. At Calvin College both the Mathematics and Physics departments are making a concerted effort to incorporate the use of Sage into their curriculums (using a Sage server run on the Computer Science department's Dahl supercomputer cluster).

Sage combines two major features. First, it provides unified interfaces to other available mathematics systems such R, PARI, and SciPy. It also features a large body of specially created modules that provide functionality not available elsewhere. In both cases, the guiding design philosophy is to focus on collaboration, rather than reinventing the wheel. Sage's custom modules are written in Python, with the focus generally on open-source maintainability and usability rather than performance.

In some cases, however, better performance is desirable. Some gains are made possible through Sage's incorporated libraries such as NumPy. Sage specific modules that require higher performance are written in Cython. Cython is a compiled language based on and compatible with Python, but with the added ability to incorporate static C typing and C functions. This generally allows Cython code to achieve significantly better performance than Python, while maintaining compatibility and accessibility.

Riemann Mapping

Before we discuss the system itself, we must examine some important background information on the mathematics of Riemann mapping. Unfortunately much of the theory is extraordinarily complex and unsuitable for a report of this nature. I will therefore explain the basic concept of a Riemann map, and give a sketch of the computationally relevant details of how the map is computed. For a more detailed explanation of the theory, as well as some information on the research that produced the initial version of riemann, the reader is advised to consult [BSV].

Conformal mapping is an important branch of complex analysis. Conformal maps map one region in the complex plane to another in such a way that angles are preserved (as well as a number of other mathematically interesting properties). Such mappings turn out to be useful in a wide variety of scientific circumstances. Moreover, both conformal mapping and a special case thereof, Riemann mapping, have fundamental significance in much of complex analysis.

A Riemann map is a conformal map from a region (let's say a square) in the complex plane to the unit circle. This means that you have a function R such that a complex number z in your square is mapped to a number R(z) on the unit circle. When R is applied to all the points in the square those points map to all the points in the circle. We then say that R maps the square to the circle.



Figure 1: The Riemann map of a square. Note how angles are locally preserved.

While the idea of Riemann mapping has been known for over a century, there are very few cases in which explicit computation of the map is possible—that is, for most regions you cannot solve for the function R in the way you would solve an algebraic equation or evaluate an integral. Instead the map must be computed numerically.

The full theoretical details of the core algorithm can be found in [KT]. I will give a sketch of the process here. The user supplies two functions, one for a parametric plot of the boundary of the region, the other a parameterization of the derivative of the region. For example, for an ellipse those functions might be:

$$f(t) = e^{it} - .5 * e^{-it}$$
$$f'^{(t)} = i * e^{it} + .5 * i * e^{-it}$$
For $0 \le t < 2\pi$

In theory, those functions are used to define other functions A and g, so that we have an integral equation:

$$\varphi(z) + \int_{w \in \partial\Omega} A(z, w) * \varphi(w) \, d\sigma_w = g(z)$$

Where φ is an unknown function that could be used to find *R*, the Riemann map.

Unfortunately, such an equation is known as an integral equation of the second kind, and it is not possible to solve directly for φ . There is however, a numerical technique known as the Nyström method that works. For Nyström, we break down the continuous functions and integral into a large number of discrete points. We evaluate the functions f and f' at a N "collocation points" around the boundary. The resulting data vectors are used to create a matrix equation that can be solved for a vector form of φ . With this, we can finally compute the value of R(z) for the collocation points.

To get R(z) for other boundary points, we can simply interpolate those collocation points. To find R(z) for points in the interior of the region, however, we must use the Cauchy Integral formula, integrating a kernel around the boundary to find the value at a point in the center.

There are two important variations on the Riemann map. First, while the Riemann map conformally maps a simply-connected region (that is, a region with no holes) to the unit circle, another map, the Ahlfors map conformally maps multiply-connected regions to the unit circle. This map is such that for a region with n holes, n + 1 points in the original region will map to 1 point in the unit circle.

Second, the exterior Riemann map conformally maps the exterior of a region to the exterior of the unit circle. Both the Ahlfors and exterior maps require only relatively trivial extensions of the algorithm that gives us the Riemann map. Examples of the Riemann, Ahlfors and exterior maps can be found in the following section.

Visualizations

The Riemann map is a function in the complex plane. Now functions in the real plane are easy to graph; one puts the variable on the x-axis, and the result on the y-axis. In the complex plane, however, one has two variables and two results—the real and imaginary parts of z and f(z). A graph of a complex function would therefore require 4 dimensions. To solve this problem, some mathematicians employ a method known as domain coloring.

Domain coloring assigns every point in the complex plane a color, thus the graph of the function is a colored image. It is read by mentally converting colors into rough numerical information. By nature then, these graphs are useful for conveying qualitative rather than quantitative information. Nevertheless, such graphs can be invaluable in understanding the macroscopic behavior of a function. Consider the example graphs below.







Figure 2 shows the colors that are assigned to each point. Notice how the positive real numbers are given shades of red, while negative reals are pale blue. Smaller numbers are darker with 0 black and infinity white. Figure 3 shows the plot of the sin function in the complex plane. Notice how along the real (x) axis, we can still see the familiar behavior of sin as it alternates from positive to negative with 0's every $\frac{\pi}{2}$.

We can adapt domain coloring to Riemann maps as well. In a colored Riemann plot, each point on the unit circle is assigned a color, and each point on the original region is colored based on where it maps to in the unit circle. Similar techniques apply to the Ahlfors and Exterior maps. Figures 4-7 provide examples of this plotting. Notice that in this case, 0 is white and infinity is black as it makes a more intuitive coloring for the figures. Notice also that the Ahlfors map has multiple points of the same color, as multiple points are mapped to the one point on the unit circle.

Riemann maps can also be plotted with a "spiderweb plot" as shown in Figure 1. These plots use the inverse Riemann map to map concentric circles and radial lines from the unit circle to the original region. This method allows certain areas of the plot to be read with more precision. Both methods have their strengths, and both are very useful in visualization of the map.



Figure 4: The colored unit circle



Figure 6: The Ahlfors map of an Annulus



Figure 5: The color plot of an ellipse



Figure 7: The exterior map of an ellipse

System Design and Implementation

I employed a highly problem driven, iterative style of development in this project. Much of the focus was on figuring out what could work, rather than planning beforehand the exact structure of the code. This methodology means that there was no monolithic design phase, nor indeed a single testing phase.

The major design decisions for this project were motivated by two major concerns: minimizing computational cost and making the tool as accessible as possible to the user. Unfortunately these two considerations are sometimes at odds.

Convention in math software systems, especially Sage, dictates certain user expectations. The user usually expects to tell the software "I have this function, graph it (or evaluate it)" in a single step. The numerical Riemann map makes that difficult. When computing the map the first step is to compute the mapping function R before one can evaluate or plot it. Now it would be possible to combine these steps so that the function is computed before each graph or evaluation. Given the cost of computing the mapping function (especially before some of the optimizations this year) and the fact that users will often want to evaluate or graph a given map multiple times, it is not desirable to compute the map several times.

Object oriented programming provides the solution. A user provides information to initialize a Riemann_Map class. This class contains all the data describing the mapping function *R* and methods that allow the user to evaluate or graph *R*, or gain access to some of its mathematically interesting data. This allows all the computation for the map to be done only once, as well as a significant amount of precomputation for evaluating and graphing to accelerate future performance. Moreover, there are already numerous elements of Sage that work this way such as the spline method that returns a function that the user can then plot or evaluate. This allows Riemann_Map to follow an interaction pattern that experienced users are already familiar with.

Effort was also made to connect as well as possible with the rest of Sage. Methods likely to have uses outside of Riemann mapping were made module rather than class level methods. In one case, the complex spline routines I created to allow mapping of arbitrary shapes were given their own module. On the flip side, I attempted minimize duplication with existing Sage features. I changed the Riemann color plotting to use as much of the existing complex plotting functionality as was efficient and feasible. This involved modifying the complex_plot module so that it would be better suited to interoperation with my and other modules.

Finally, much of the internal design of the module is dictated by the constraints of the language. For example, function calls are very expensive in Python, and Cython. Accordingly I attempted to employ relatively little method extraction in important parts of the code that are called frequently during normal use. In addition, Cython is still a work in progress and certain desired features are not yet available. In some cases this dictated design decisions. Most

importantly, Cython does not support full typing of class variables. Therefore significant data is copied to function variables where computations can be performed more efficiently.

Actual implementation of the upgrades followed a two step process. First, I worked on an individual upgrade, using git for internal version control. Once progress reached satisfactory levels, the Sage Mercurial version control system produced a patch that was uploaded to the Sage trac server. In the second stage the Sage community reviewed the patch—downloading and testing it on multiple platforms—and provided feedback on usability and readability of the code. In some cases these steps were reversed; members of the community created tickets highlighting issues that they found in the current version of the riemann module.

Parallelization

Initial work focused on optimizing and parallelizing the module. Most of the first semester was devoted to exploring the options for parallelization. Working within the Sage framework limited the possibilities. It was infeasible to add any new major libraries such as mpi4py to the Sage distribution, merely for one module. Fortunately, Python provides some robust parallelization options, although the overhead is large.

At the start of development, there were two computational bottlenecks. First, the initial evaluation of the boundary function was very costly. Although it scaled linearly with the number of points, for most use cases it was the dominant computational cost. This bottleneck was easily parallelized. However, at the same time, working with a member of the Sage community, riemann was converted to use Sage's fast_callable functionality. This dramatically reduced the cost of this bottleneck, rendering the additional overhead and complexity of parallelization unnecessary.

The second bottleneck, computing the color plot of a map, was also reduced using nonparallel methods, although it remains the most computationally intensive section of code. Parallelization could be added, and it is possible that in the future that will be advantageous. At the moment, however, Sage is more focused on functionality and accessibility. Indeed very few parts of Sage use parallel algorithms. In addition, the performance of the plot is entirely consistent with the standard latencies encountered in Sage. Therefore, since the primary uses of this software are not especially high-performance, the added complexity and reduced accessibility to the open source community make parallelization undesirable at this stage.

Algorithms

Most of the algorithms used in this project are either simple or documented in [BSV] and [KT]. However, the algorithm developed to compute spiderweb plots for multiply-connected regions is original and significant. I will therefore discuss the algorithm and its justification in the following section.

As discussed in Background, the spiderweb plot (Figure 1) is a useful visualization tool, revealing information about the behavior of a Riemann map that isn't always apparent from a color plot. For simply-connected domains the inverse Riemann map is used to map the points of the spiderweb from the unit circle to the region. For multiply connected domains (such as the annulus in Figure 6) the mathematics underlying the inverse map breaks down and this technique becomes unusable.

Instead of using the inverse map, the regular Riemann map can be used. We employ a method similar to color plotting. Each point on the figure is mapped to the unit circle. However, instead of assigning it a color based on domain coloring, we color the point if it falls on one of the lines or circles of the spiderweb, and leave it blank otherwise. This straightforward solution is as computationally intensive as a color plot and gives us a graph such as Figure 8.



Figure 8: The naive spiderweb plot

This graph resembles a spiderweb plot (note that there are two spiderwebs because this is a 2 to 1 Ahlfors map), however there is substantial distortion in the large black regions. This is because we are not mapping our points to perfect mathematical lines or circles, but rather to finite rectangles and rings. Those regions can get stretched by the map in the same way that certain shades of red and blue are stretched in Figure 6. Consider how the algorithm works. It takes a point, maps it to the circle and checks if it hits one of the target rectangles or rings. It then takes an adjacent point and maps it to the circle. In some parts of the region, the map changes very slowly, thus many points get mapped very close together, in the same target region, resulting in them all being colored.

Our problem then is that the size of our targets can be exaggerated when the map is changing very slowly. To solve this, the size of the targets must take into account the rate of change of the map. In other words when the derivative of the map is small, the target must be small. Therefore, we need to know the derivative of the map at all of the plotted points. Now it turns out that the same techniques that give us the Riemann map can also give us the derivative. However, that derivative would have to be computed at every point using the Cauchy integral which usually works out computationally as taking the dot product of two length 500 vectors for each of our N * N plotting points (N = 400 typically gives a good looking plot). We're already doing this once for the value of the map, and we'd prefer not to have to do it a second time.

Using a technique called Taylor approximation, we can use the value data we already computed to get a very good approximation of the derivative. The derivative of a function f can be approximated by this equation:

$$f'(x) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{\Delta x}$$

Or, since we're using complex numbers by the following two equations. f_x is the rate of change in the real direction and f_y is the change in the imaginary direction.

$$f_x(x+iy) \approx \frac{f(x+\Delta x+iy) - f(x-\Delta x+iy)}{\Delta x}$$
$$f_y(x+iy) \approx \frac{f(x+i(y+\Delta y)) - f(x+i(y-\Delta y))}{\Delta y}$$

How do we implement this? Let *R* be the matrix with the Riemann value data that we have already computed with spacing of Δx and Δy . We can then determine two more matrixes:

$$R_{x} = \frac{R[1:-1,2:] - R[1:-1,:-2]}{\Delta x}$$
$$R_{y} = \frac{R[:-2,1:-1] - R[2:,1:-1]}{\Delta y}$$

This gives us approximate values for the derivative in the real and imaginary directions. ¹Not only that, we've done it with one addition and one division per point per direction, rather than 500 multiplications. To use these derivatives to adjust our target sizes, we want to convert them to polar coordinates. Then the rate of change of radius can adjust circle precision and rate of change of angle can adjust line precision. We can find the rate of change in the *r* and θ directions as a linear combination of the x and y derivatives. This requires solving a small matrix equation for both *r* and θ . The net effect of setting up and solving this equation is a fairly significant computational burden.

¹ The resulting matrices are 1 point smaller in each direction, but this makes no practical difference in a plot of dimension 400 x 400. Those boundary points are simple ignored; they're usually outside of the interesting region anyway.

Fortunately there's a better way. Recall from the beginning that the Riemann map is a "conformal" map. This means that it is an analytic function which, among other things, means that it satisfies the Cauchy-Riemann equations. This in turn means that at any given point, the magnitude of the derivative is the same in all directions. All that is to say that we don't have to solve a matrix to find the magnitude of the *r* derivative because it's *the same* as the x and y derivatives. Moreover, we don't even need to find the y derivative, all we need is the x derivative. Thus we can get the two derivatives of interest as follows, where the $\left|\frac{1}{R}\right|$ term converts from velocity to angular velocity.

$$|R_r| = \left| \frac{R[1:-1,2:] - R[1:-1,:-2]}{\Delta x} \right|$$
$$|R_{\theta}| = \left| \frac{R[1:-1,2:] - R[1:-1,:-2]}{\Delta x} \right| * \left| \frac{1}{R} \right|$$

Finally, we define target radii and angles based on the number of circles and lines in the spiderweb and test $|radius - target| < \frac{precision}{R_r}$ and $|angle - target| < \frac{precision}{R_{\theta}}$ for each point. We can define *precision* based on the user specified line thickness. As Figure 9 shows, immediately outside the boundaries the small but non-zero numerical derivative throws off the target size, but adding a minimum value and maximum R_{θ} eliminates it, leaving us with the perfect spiderweb plot in Figure 10.



Figure 9: Boundary Fuzz

Figure 10: A clean spiderweb plot

Testing

Sage requires substantial doctests for all of its modules. Test cases are written in the documentation for each class and method, using the syntax of an example use case. For example, the following code tests the class method that returns the value of the Riemann map at a given point.

```
sage: f(t) = e^(I*t) - 0.5*e^(-I*t)
sage: fprime(t) = I*e^(I*t) + 0.5*I*e^(-I*t)
sage: m = Riemann_Map([f], [fprime], 0)
sage: m.riemann_map(0.25 + sqrt(-0.5))
(0.137514...+0.87669602...j)
```

The first four lines are Sage code that is evaluated by the testing function, the last line is the expected output, a complex number (NumPy syntax uses a + bj rather than a + bi). These tests serve the dual purpose of providing automated testing akin to unit testing, and providing example use cases for users.

For this project, there are two major issues that arise in testing, usability testing, and numerical testing. Usability testing is particularly as the target user is one who is both familiar with Sage, and with the mathematics of Riemann mapping. Such users are in short supply, making rigorous testing difficult. Instead substantial feedback from the Sage community helped shape usability. This feedback both explicitly recommended certain improvements in the documentation and highlighted places where the documentation was unclear, leading to misunderstandings.

The second issue, numerical testing, arises from the fact that this project revolves around advanced numerical mathematics. First, this means since a tool on the order of this project is necessary to do most Riemann mapping, some valuable test cases have no canonical value for comparison. Second, the algorithms used in this project rely heavily on functions from Sage's numerical libraries. These functions are numerically sensitive; their exact output is potentially platform, OS, and version dependent. As a result, doctests that succeed on an AMD Linux machine may fail under OSX and again when Sage updates to the next version of NumPy. Fortunately these failures are only numerical fuzz, with the differences less than the precision of the numerical calculations. This fuzz can be safely ignored in test cases using the "…" syntax seen in the example above.

Given these precision problems, how do we know the code is performing correctly? First, a visual check on the plots can reveal most abnormalities and check for conformality (right angles in the spiderweb plot). Second, there are a few figures for which an explicit map is known. riemann contains a set of methods that compute the explicit map for the ellipse to a high degree of precision and use it to test the reliability of the normal numerical mapping technique.

Results

Overall, this project produced significant performance gains, particularly in the initial computation of the map, as shown in the graph below. For 500 collocation points, a normal use case, we see roughly 15x speedups. For 1000 points, a high-precision case, speedup is roughly 8x. As the number of points increases the advantage vanishes because the primary cost becomes solving the Nyström matrix equation which relies on the LAPACK library.



In addition to these optimizations, The riemann module now has the capability to compute the exterior map and the Ahlfors spiderweb plot reliably and efficiently. Moreover, the code has been improved in many places, improving tests, reliability, readability and integration with Sage as a whole.

Future Work

The project stands completed and will be included in future Sage releases. The Sage review process operates on its own schedule, however, and it may take a couple of months before all the publishing kinks are worked out. I intend to stay active in the maintenance of this project through its publication.

In addition to the publishing work, there are a few other areas where profitable future work may be done. First, while the Sage community usability feedback is helpful, a more complete usability study would be profitable. In the next few years as Calvins integration of Sage into the curriculum progresses, the number of students with both the Sage experience and knowledge of upper level math necessary to be informed users will increase dramatically. It may be feasible for a future student to enlist Complex Analysis students to do rigorous usability testing.

There is also room for future optimization. As Sage evolves, it may become more feasible and beneficial to parallelize the algorithms. In addition, there may be room for substantial improvements to the performance of the color plotting function. Profitable avenues may include dynamic precision—fewer on the easy interior and more near the mathematically unstable boundary. It might also be advantageous to include interior-exterior recognition, allowing the algorithm to avoid unnecessary computations for points outside of the figure.

Finally, this project opens up avenues for future mathematical work. The riemann tool can help future theoretical exploration and research. Other types of conformal mapping capabilities could also be added to this module.

Conclusions

Working on this project taught a great deal. First, I gained experience working with highly numerical code. I now have a deeper intuitive understanding of how numerical algorithms work, and some of the problems of implementation that must be addressed. Similarly, I have also gained valuable experience working as part of an open source community. The experience working within Sage's collaborative, iterative framework has proven invaluable in broadening my programming horizons. I would recommend that the CS department makes an effort to encourage CS students to participate in such open source projects. The different types of problems and techniques that one is exposed to are potentially very valuable to new programmers.

Finally, this project illustrates some of the strengths of a combined background in mathematics and computer science. A heavily mathematical project such as this requires a fair degree of mathematical expertise. More importantly, the Ahlfors spiderweb algorithm shows how these two fields combine. Computer Science allows the implementation of the algorithm, and identifies the advantages of using pre-computed data, rather than taking the purely mathematical route of specially computing all the derivatives. Mathematics on the other hand, shows why the derivatives are necessary in the first place. Numerical analysis, a branch of mathematics commonly used in computer science, provides the Taylor approximations used to compute derivatives. However, it takes complex analysis and the Cauchy-Riemann equations, mathematics that is typically classified as pure, to reveal the final trick that makes the computation of the spiderweb efficient. The synergy of these two fields is something that I intend to profit from in my future endeavors.

Acknowledgements

I would like to thank my advisors, Professor Joel Adams for his excellent guidance in the computer science of this project, and Professor Michael Bolt for his invaluable support both in this honor project and the 2009 research.

I would also like to thank the Sage community for all of the help. Special thanks go out to Jason Grout for his work on the fast_callable functionality and François Bissey and Karl-Dieter Crisman for their help improving the overall package.

References

Papers

- [KT] N. Kerzman and M. R. Trummer. "Numerical Conformal Mapping via the Szego kernel". Journal of Computational and Applied Mathematics, 14(1-2): 111--123, 1986.
- [BSV] M. Bolt, S. Snoeyink, E. Van Andel. "Visual representation of the Riemann map and Ahlfors map via the Kerzman-Stein equation". Involve 3-4 (2010), 405-420.

Trac Tickets

- **#8867** "speed up the riemann mapping functionality". <u>http://trac.sagemath.org/sage_trac/ticket/8867</u>
- #10792 "Upgrade numpy to 1.5.1". http://trac.sagemath.org/sage_trac/ticket/10792
- **#10821** "riemann.pyx gives lots of invalid value in divide warnings". <u>http://trac.sagemath.org/sage_trac/ticket/10821</u>
- **#10945** "Fix lots of minor docs and redundancy for riemann.pyx". <u>http://trac.sagemath.org/sage_trac/ticket/10945</u>
- #10957 "Add analytic testing to riemann.pyx". http://trac.sagemath.org/sage_trac/ticket/10957
- #11028 "More Modular ComplexPlot". http://trac.sagemath.org/sage_trac/ticket/11028
- **#11273** "Riemann Enhancements: Docs, Exterior, Multiple Spiderweb, Error Testing". <u>http://trac.sagemath.org/sage_trac/ticket/11273</u>

Sage Website

http://www.sagemath.org/

Appendix: riemann.pyx Source Code

Up to date versions of this code will be available from the Sage website by fall of 2011.