Amoeba Program: Computing and visualizing amoebas for some complex-valued bivariate expressions

Magnus Leksell
Wojciech Komorowski
October 2007

Bachelor’s Degree in Mathematics
Supervisor/examiner: Mikael Forsberg
Amoeba Program: Computing and visualizing amoebas for some complex-valued bivariate expressions

by

Magnus Leksell
Wojciech Komorowski

Department of Mathematics, Natural- and Computer science,
University-College of Gävle

S-80176 Gävle, Sweden

E-mail:
nfk03mll@student.hig.se
nmd04wki@student.hig.se

Abstract

In this paper, we present an implementation of a C++ program for visualizing the amoebas of arbitrary bivariate polynomials with complex coefficients, power series and simple bivariate functions of the form \( f(z, w) = z + \cos w \).

Keywords: Amoeba, Polynomials, Complex numbers, Arbitrary precision, Parallel computing, C++, XML.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>2</td>
</tr>
<tr>
<td><strong>1</strong> Introduction</td>
<td>3</td>
</tr>
<tr>
<td><strong>2</strong> Program requirements</td>
<td>3</td>
</tr>
<tr>
<td><strong>3</strong> Implementation</td>
<td>4</td>
</tr>
<tr>
<td>3.1 The polynomial</td>
<td>4</td>
</tr>
<tr>
<td>3.2 The intervals</td>
<td>4</td>
</tr>
<tr>
<td>3.3 Parallel computing</td>
<td>5</td>
</tr>
<tr>
<td>3.4 Root-finding</td>
<td>5</td>
</tr>
<tr>
<td>3.5 The rendering</td>
<td>8</td>
</tr>
<tr>
<td>3.5.1 The Rendering of Amoebas</td>
<td>8</td>
</tr>
<tr>
<td>3.5.2 The Rendering of The Newton-polytope</td>
<td>9</td>
</tr>
<tr>
<td>3.6 XML parsing</td>
<td>9</td>
</tr>
<tr>
<td><strong>4</strong> Tests</td>
<td>10</td>
</tr>
<tr>
<td>4.1 Root-finder test</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Program tests</td>
<td>12</td>
</tr>
<tr>
<td><strong>5</strong> Discussion</td>
<td>21</td>
</tr>
<tr>
<td>5.1 The result</td>
<td>21</td>
</tr>
<tr>
<td>5.2 About the work</td>
<td>21</td>
</tr>
<tr>
<td>References</td>
<td>22</td>
</tr>
<tr>
<td><strong>A</strong> Description of command line options</td>
<td>23</td>
</tr>
<tr>
<td><strong>B</strong> The structure of the XML config file</td>
<td>24</td>
</tr>
<tr>
<td><strong>C</strong> Taylor tests</td>
<td>26</td>
</tr>
<tr>
<td><strong>D</strong> Source code</td>
<td>29</td>
</tr>
</tbody>
</table>
Preface

This paper is part of the Bachelor’s Diploma Work in Mathematics by Magnus Leksell and Wojciech Komorowski at the Department of Mathematics, Natural- and Computer science, University-College of Gävle.

The work is about making the program Amoeba Generator, which is developed using C++ and some open source libraries such as: stl – the Standard Template Library, CLN for arbitrary precision numbers, GD for image creation and libxml++ for XML parsing.

Except parts of the work and code that we developed together, there are some parts of the work which one can say was done mostly from one of us, and these are:

**Magnus Leksell**  Overall system design, parallel processing/computing, XML parsing, power series, polynomial representation, \LaTeX-producing, initial image creation code and administration of the Subversion repository.

**Wojciech Komorowski**  Maybe the most important part of the program; implementation of the root-finding algorithm, amoeba and Newton polytope image rendering and command line parsing.
1 Introduction

The word Amoeba is a recent addition to the mathematical terminology, introduced by I. M. Gelfand, M. M. Kapranov and A. V. Zelevinsky in their book “Discriminants, resultants, and multidimensional determinants” from 1994 [GKZ]. It is defined as the image of the zero locus of a multivariate polynomial under the map

$$\text{Log} : (\mathbb{C} \setminus 0)^n \rightarrow \mathbb{R}^n : (z_1, \ldots, z_n) \rightarrow (\log |z_1|, \ldots, \log |z_n|),$$

where 'log' denotes the natural logarithm. For the case of bivariate polynomials, the amoeba

$$\mathcal{A}_p = \{\text{Log}(z) : z \in (\mathbb{C} \setminus 0)^2, p(z) = 0\}$$

is planar.

Planar amoebas of a not identically zero bivariate polynomial have a finite area. If the bivariate polynomial is not identical to any univariate polynomial, then the amoeba is 2-dimensional and has a number of infinitely long and narrowing “tentacles”, and can even have holes (vacuoles), which makes it resemble a biological amoeba.

Every amoeba is associated with its Newton-polytope, which for a 2-dimensional amoeba is the convex hull of the exponents of the monomials of the amoeba’s polynomial. As an example, the Newton-polytope of the polynomial $p(z, w) = 1 + z + w$ is the polygon with vertices at (0, 0), (1, 0) and (0, 1). Relations between Newton-polytopes and amoebas have been discovered by Mikael Forsberg, Mikael Passare and August Tsikh [FPT]. One such relation is that an amoeba’s tentacles are always perpendicular to the sides of its Newton-polytope.

Amoebas are a relatively fresh field of research, and as far as we know, there does not exist any program with the general purpose of visualizing them.

Our objective has been to construct such a program, capable of visualizing amoebas of bivariate polynomials, power series and simple bivariate functions of the form $f(z, w) = z + \cos w$.

2 Program requirements

The requirements was to create a program that can:

- Calculate the amoeba for a polynomial in at least two variables.
- Calculate the amoeba for a power series in at least two variables.
- Calculate the amoeba for a function of type $z + \cos w$.
- Plot lists of real number pairs.
- Calculate and plot Newton-polytopes.
- Generate information about the polynomial (and the power series) in \LaTeX.
3 Implementation

The program works by first parsing the command line for input. Then, either parse a XML-configuration file, setting up some initial parameters in the Amoeba class and, for each interval, compute the amoeba and produce images (the amoeba and the Newton polytope) and a $\text{\LaTeX}$-file. Or plot an image from values given in a file. That are the two main functionalities of the program. In the XML-file one sets up what to be computed, a bivariate polynomial, power series or, a bivariate expression of type $z + \cos w$. Note that $\cos w$ will be approximated with a Taylor polynomial in desired terms (default 10 terms).

The computing of the zeros of the given polynomial is divided into intervals (or regions), and as the zeros are found for each region, they are saved in a file for later use. In this way, computing the amoeba is very memory efficient.

All error management is implemented using C++ exceptions and specifically the stl’s exceptions runtime_error and out_of_range.

3.1 The polynomial

The bivariate polynomial is represented in C++ by the stl container map, as

```cpp
typedef std::map<Key, cln::cl_N> CPolynomial;
```

where `Key` is defined as typedef pair<int, int> Key and `cl_N` is a complex number class. For example, the polynomial $2i - z^2 + zw^3$ would be represented as:

```cpp
def poly{0,0} = cln::complex(0,2); // (0+2i)z^0w^0
def poly{2,0} = cln::complex(-1,0); // (-1+0i)z^2w^0

def poly{1,3} = cln::complex(1,0); // (1+0i)z^1w^3
```

This is a very simple but still a very powerful structure to represent a bivariate polynomial, and is used throughout the whole program. We also use the CLN structure cl_UP_N, which is a univariate polynomial with complex coefficients, when constructing the $p_z$ and $p_w$ polynomials respectively (which is more in depth described in section 3.5.1).

3.2 The intervals

The calculation of the amoeba is divided into intervals, which will make computations a lot faster, and one can also do some experiment more precisely. These intervals are defined in the XML-configuration file and are implemented as two classes, Square and Circle. These two classes inherits an abstract base class, AbstractInterval, which is one of the input parameter types to the method compute in the class Amoeba.

The abstract base class represents a region in the complex plane from which substitute values will be taken to replace one of the variables in the bivariate polynomial $p(z,w)$, so that univariate polynomials can be created as a step in plotting the amoeba.

To determine what kind of interval which will be used in the compute method, we use the C++ function typeid, as the following sample code shows.

```cpp
void Amoeba::compute(const CPolynomial& p, const AbstractInterval* interval, sem_t* mutex) {
    // Determine the type of interval
    if (typeid(*interval) == typeid(Square)) {
```
3.3 Parallel computing

Computing the amoeba is done using parallel processes, which is beneficial for architectures with more than one core/CPU. Tests on a dual-core machine show that time spent on computing, using two parallel processes, is cut down by 50% compared to using only one process. And one can expect that more cores or CPU’s will cut down computing times even further. That is, the number of parallel processes should equal the number of CPU cores.

Due to the fact that the [CLN] library is not thread safe, we were forced to implement parallelism using fork() instead of threads, which might have been a more modern choice. But after some thinking we came up with a clean solution. The computed data for each interval is saved in a file, and to prevent data corruption, each process locks the file before saving the data. The file locking mechanism is implemented using a semaphore variable (mutex), which is allocated in a shared memory area, which enables it to be shared between the processes. The main computing algorithm in pseudo code is listed in algorithm 1.

**Algorithm 1** Pseudo code for the main computing process

```plaintext
1: interval ← first interval
2: processCount ← 0
3: while interval ≠ end(interval) do
4:   create process and compute amoeba for interval
5:   processCount ← processCount + 1
   // If maximum number of processes running, wait for any process to finish
6:   if processCount == MAX_PROCESSES then
7:     wait for a process to finish
8:     processCount ← processCount - 1
9:   end if
10: interval ← next(interval)
11: end while
   // Wait for remaining processes to finish
12: while processCount > 0 do
13:   wait for a process to finish
14:   processCount ← processCount - 1
15: end while
```

3.4 Root-finding

We have evaluated several algorithms for finding roots of univariate polynomials with complex coefficients.
Among them was the Newton-Rhapson method, which has quadratic convergence for simple roots and linear convergence for multiple roots. It did however not satisfy our needs since, in a straight forward implementation, polynomial deflation has to be used in order to find all the roots of a polynomial. Polynomial deflation works by finding one root of the polynomial and factoring it out by polynomial division, leaving a new polynomial of a smaller degree, for which the algorithm is applied again, until all the roots have been found. There are several problems with this approach.

Despite working with arbitrary precision arithmetic, there are inevitably smaller errors introduced with every arithmetic operation. These errors would grow in the coefficients of the polynomial obtained after each polynomial division.

Furthermore, the problem of finding roots of arbitrary polynomials has been shown to be ill-conditioned [Wilkinson], in the sense that small perturbations in the coefficients of a polynomial can lead to large perturbations in the roots of the polynomial.

Consider Wilkinson’s polynomial: \( w(x) = (x - 1)(x - 2) \ldots (x - 20) \), which clearly has 20 integer roots. Expanding the polynomial gives:

\[
\begin{align*}
    w(x) &= x^{20} - 210x^{19} + 20615x^{18} \\
    &- 1256850x^{17} + 53327946x^{16} \\
    &- 1672280820x^{15} + 40171771630x^{14} \\
    &- 75611184500x^{13} + 1131027695381x^{12} \\
    &- 1355851828999530x^{11} + 1307535010540395x^{10} \\
    &- 10142299865511450x^9 + 63030812099294896x^8 \\
    &- 31133643161390640x^7 + 1206647803780373360x^6 \\
    &- 3599979517947607200x^5 + 8037811822645051776x^4 \\
    &- 12870931245150988800x^3 + 13803759753640704000x^2 \\
    &- 8752948036761600000x + 2432902008176640000.
\end{align*}
\]

If the coefficient of \( x^{19} \) is decreased from \(-210\) to \(-210.0000001192\), then the root at \( x = 20 \) grows to \( x \approx 20.847\) [ratfun].

This does not bode well for a root finding algorithm based on polynomial deflation.

Another problem with the Newton-Rhapson method is that the set of initial values in the complex plane for which the algorithm converges to a root is a fractal depending on the polynomial. Some convergence fractals have regions where initial values never converge. To solve this, one could take a heuristic approach, letting an initial value jump around until the algorithm converges to a root, however that does not solve the problems mentioned earlier.

In a paper by John Hubbard, Dierk Schleicher and Scott Sutherland [HSS], a method of finding all roots of complex polynomials by Newton’s method with no intermediate deflation is presented. We did not chose this path because we felt an implementation would be complex and hard to debug.

Another popular algorithm we considered was the Jenkins-Traub algorithm [JT], [RR], which has been described as “practically a standard in black-box polynomial root-finders” [Press]. It is an iterative three-stage method with faster convergence than the Newton-Rhapson method. The algorithm is pretty complicated, making an implementation hard to debug. Unfortunately it is also based on polynomial deflation, and
since we prioritize accuracy over speed when it comes to root finding, we chose not to implement it.

We also considered a method from linear algebra: Given the polynomial

\[ p(z) = a_n z^n + a_{n-1} z^{n-1} + \cdots + a_1 z + a_0, \]

with \( a_n = 1 \), one can form its \( n \times n \) companion matrix

\[
C = \begin{pmatrix}
0 & -a_0 \\
1 & -a_1 \\
0 & 1 & -a_2 \\
& \ddots & \ddots \\
& & 1 & -a_{n-1}
\end{pmatrix}
\]

with characteristic polynomial \( P_C(z) = \det(zI - C) = p(z) \). The problem of finding the roots of \( p(z) \) is thus equivalent to computing the eigenvalues of \( C \) with any suitable algorithm. We rejected this method because:

- The companion matrix requires \( O(n^2) \) storage, which becomes a problem for polynomials of higher degrees
- The need of implementing an algorithm for finding the eigenvalues
- The accumulated errors in the eigenvalues after executing such an algorithm.

After having reviewed and rejected many more algorithms for similar reasons, we finally settled with the Weierstrass or Durand-Kerner method [Petkovic-Weierstrass]. It is an incredibly simple iterative algorithm with quadratic convergence to simple roots and linear convergence to a multiple root. The algorithm finds all roots of a polynomial simultaneously and without the need of polynomial deflation.

Given a monic polynomial \( P \) of degree \( n \geq 3 \), the method works by setting up a vector \( Z = (z_1, \ldots, z_n) \) of distinct trial roots to \( P \). Each \( z_i \) in \( Z \) is then refined by letting \( z_i^{\text{new}} = z_i^{\text{old}} - W(z_i^{\text{old}}) \), where \( W \) is Weierstrass’ correction defined as

\[
W(z_i) = \frac{P(z_i)}{\prod_{j=1, j \neq i}^{n} (z_i - z_j)}.
\]

By successive iterations of these refinements, \( Z \) will eventually converge to the vector \( R = (r_1, \ldots, r_n) \) of roots to \( P \).

In pseudo code, the algorithm for a general \( n \):th degree polynomial \( P \) is listed in algorithm 2.

There is some loss of precision in making a given polynomial monic, however it is not nearly as bad as repeatedly deflating it.

The speed at which the algorithm converges to the roots is dependent on the initial trial roots, and a natural question to ask is therefore which trial roots to use. We have found by experiments that distributing them around the unit circle is a good approach, and it is what our implementation uses.

The convergence check in our implementation is done by comparing \( |P(z)| \) for each \( z \) in \( Z \) to a user defined error value. We have chosen to loop part 3. of the pseudo
Algorithm 2 General $n$:th degree polynomial

1: Make $P$ monic (i.e. make the leading coefficient a 1 without changing the zero set of the polynomial)
2: Setup the starting vector $Z = (z_1, \ldots, z_n)$
3: for $i = 1$ to $n$ do
4: \[ Z[i] = Z[i] - P(Z[i]) / \prod_{j=1, j \neq i}^{n} (Z[i] - Z[j]) \]
5: end for
6: if no convergence, goto 3. (or give up)

code $n$ times for an $n$:th degree polynomial before checking for convergence, and then in steps of $n$ times until convergence occurs. Empirical tests showed this to be a good approach. We have included a threshold of 65535 iterations, should the algorithm fail to converge. In such a case, it will print an error message on stderr and continue.

Failure of convergence for this method is extremely rare. We have only come across it when specifying “unrealistic” error bounds, for which a given working precision simply is not enough. In each case, however, the error bounds could be satisfied by working with a higher precision.

It should be noted, unfortunately, that it is always possible to construct a polynomial which defeats any particular root finding algorithm working with a predefined precision. Polynomials of very high degrees, or which have roots with a huge difference in magnitude, or roots which are very tightly clustered fall into this category.

Test runs on various polynomials and amoebas are provided in section 4.

3.5 The rendering

When constructing the image, the program first reads all the points from the data-file that was created when calculating the zeros of the polynomial (or power series), and then calculates and adjust the points to fit an image structure. All images are created using the GD library and we chose the image type to be png.

3.5.1 The Rendering of Amoebas

Our approach to rendering the amoeba of a bivariate polynomial $p = p(z, w)$ requires constructing univariate polynomials $p_z(w)$ and $p_w(z)$ from $p$. This is done by replacing one of the variables of $p$ with constants. These constants come from intervals, or regions in the complex plane, specified in the user-provided XML-file.

The regions have the shape of a rectangle, or a disk centered at the origin in the complex plane with a hole in the middle (much like a CD or DVD disk). Each region has a finite amount of points contained within them. For more information about these regions, see section 3.2.

An outline of the amoeba rendering algorithm is given below:

1. Setup a list that will contain points that make up the amoeba. We can call it the $A$-list.
2. For each point of each region:
   2.1 Construct $p_z(w)$ and $p_w(z)$ by replacing $z$ and $w$ with the point, respectively.
2.2 Set up two lists that will hold the roots of $p_z(w)$ and $p_w(z)$. We can call them $p_z(w)$-roots and $p_w(z)$-roots.

2.3 Find all the roots of $p_z(w)$ and $p_w(z)$ and place them in their respective roots-list.

2.4 For each root in $p_z(w)$-roots:

2.4.1 If $\log|z|$ and $\log|\text{root}|$ are within the limits of $\min\ln Z/\max\ln Z$ and $\min\ln W/\max\ln W$ respectively, then construct the point $(\log|z|, \log|\text{root}|)$ and place it in $A$-list.

2.5 For each root in $p_w(z)$-roots:

2.5.1 If $\log|\text{root}|$ and $\log|w|$ are within the limits of $\min\ln W/\max\ln W$ and $\min\ln Z/\max\ln Z$ respectively, then construct the point $(\log|\text{root}|, \log|w|)$ and place it in $A$-list.

3. Create an image with the dimensions specified in the XML-file, and map all points in $A$-list to it.

The mapping of points in the amoeba to the image in step 3. is such that the whole width and height of the image is used. The mapping is therefore appropriately scaled to make full use of the image dimensions.

If one chooses to use axes in the image, then the horizontal axis correspond to $\log|z|$ with positive direction to the right, and the vertical axis corresponds to $\log|w|$ with positive direction upwards.

For more information about how the roots of $p_z(w)$ and $p_w(z)$ are found, see section 3.4.

3.5.2 The Rendering of The Newton-polytope

The Newton-polytope to $p(z, w)$ is constructed from the exponents of the monomials of $p$ with non-zero coefficients. Every such monomial is represented as a small disk, or dot, in the image. The location of each dot is dependent on the exponents of the monomial. Increasing exponents in the $z$ variable of the monomial pushes the dot further to the right, and increasing exponents in the $w$ variable of the monomial pushes the dot further upward.

The convex hull is constructed using the Graham’s Scan algorithm [O’Rourke]. The dimensions of the image are determined automatically.

3.6 XML parsing

Implementation of the parsing of the configuration-file was done using the libxml++ library, which is a C++-wrapper for the libxml library. It was fairly complicated to implement XML-parsing functionality, but it was absolutely worth it, as it allows us to expand and/or extend the functionality of the program more easily.

One small limitation of the structure in the XML-configuration file is that the $<\text{settings}>...</text{settings}>$ entity should be declared in the head of the file. That is because the $<\text{max precision="n"/>$ is used when constructing the other numbers. If one does not declare the $<\text{max precision="n"/>$ at all, the default precision (50) is used.


4 Tests

What follows are some tests. We intentionally did not include any tests of the \texttt{-p, -plot=LISTFILE} command, because we know it is working and it is not much to test. Also, most of the tests produces a \LaTeX\-file which is not shown in these tests. See also Taylor tests in appendix C.

4.1 Root-finder test

Since drawing amoebas by our method depends explicitly on the ability of the polynomial root finder to perform its task, we have extensively tested it with satisfactory results. We have constructed a small program called “testroots”, specifically for this purpose. The test program reads a file that defines which polynomial the root finder shall try to solve, and presents the result. The root finder used in the tests differs only slightly from the one used in the main program, the only difference being that the one in the main program does not report the number of iterations used to solve a polynomial.

The testroots program, along with various test polynomials, can be found in the amoeba/source/polynomial/rootfinder directory. The README file therein present more information about the format of the files used to define polynomials to test. The output generated by the testroots program can be pretty huge. We have therefore chosen not to include all tests directly in this report. Instead, we choose to include only the output of testroots on Wilkinson’s polynomial, with a working precision of 50 decimal digits and an error threshold of 1e-30.

\begin{equation}
    w(x) = x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + 53327946x^{16} - 1672280820x^{15} + 40171771630x^{14} - 756111184500x^{13} + 11310276995381x^{12} - 1355851828999530x^{11} + 1307535010540395x^{10} - 10142299865511450x^9 + 63030812099294896x^8 - 311333643161390640x^7 + 1206647807380737360x^6 - 3599979517947607200x^5 + 8037811822645051776x^4 - 12870931245150988800x^3 + 13803759753640704000x^2 - 8752948036761600000x + 2432902008176640000.
\end{equation}

below:

\$ ./testroots testpolynomials/wilkinson20

\begin{verbatim}
p(x) = 1.0L0+0.0L0ix\^20 + -210.0L0+0.0L0ix\^19 + 20615.0L0+0.0L0ix\^18 + -1256850.0L0+0.0L0ix\^17 + 5.3327946L7+0.0L0ix\^16 + -1.67228082L9+0.0L0ix\^15 + 4.01717716L3L10+0.0L0ix\^14 + -7.56111184L5L11+0.0L0ix\^13 + 1.13102769L5381L13+0.0L0ix\^12 + -1.35585182L89953L14+0.0L0ix\^11 + 1.30753501L0540395L15+0.0L0ix\^10 + -1.01422998L6551145L16+0.0L0ix\^9 + 6.30308120L99294896L16+0.0L0ix\^8 + -3.11333643L16139064L17+0.0L0ix\^7 + 1.20664780L37336L18+0.0L0ix\^6 + -3.59997951L79476072L18+0.0L0ix\^5 + 8.037811L1822645051776L19+0.0L0ix\^4 + -1.28709312L45098888L19+0.0L0ix\^3 + 1.38037597L53640704L19+0.0L0ix\^2 + -8.75294803L67616L18+0.0L0ix + 2.43290200L817664L18+0.0L0i
\end{verbatim}
Total iterations: 1580
roots are:
r1:
18.99999999999999999999999999999999999999999997747808080741L0
3.341623260363450031832024267949101804870153910039777739L-2824i
r2:
10.000000000000000000000000000000000000000000000212899034792L0
3.47057591235824682751716401424411749539254085888924835862L-2824i
r3:
13.99999999999999999999999999999999999999999999216463252234L0
3.406988727118285830634969586701756066157721449964138234156L-2822i
r4:
16.0000000000000000000000000000000000000000000003023232904L0
-2.0640834986331288156459216928037186609839236662317L-2843i
r5:
17.00000000000000000000000000000000000000000000064042038428L0
6.233344399405880763632123521474637323495517632763958683L-2841i
r6:
18.00000000000000000000000000000000000000000000012541536247L0
8.42994760995633116861425555783319578933469779815735327317L-2844i
r7:
19.9999999999999999999999999999999999999999999945582868561L0
-1.50895479672486531504995161054138389326582987037636903934L-2845i

r9:
12.000000000000000000000000000000000000000000000531255729008L0
-3.345328725896412249391171074936361703022971660271196270191L-2843i
r10:
5.0000000000000000000000000000000000000000000002571422L0
-9.537664928214108585109116583134946145137417921135706522L-2854i
r11:
15.000000000000000000000000000000000000000000000114339703464L0
9.447914503218040947936762990999625573050166503179968742L-2848i
r12:
12.9999999999999999999999999999999999999999999958463593557L0
-4.910680794183162198029036150532519087130729193087554107469L-2862i
r13:
10.99999999999999999999999999999999999999999999638246992138L0
6.423641755237283616668063699144995787897123457819035583991L-2867i
r14:
9.0000000000000000000000000000000000000000000008202188366L0
3.26318312402243370399416423308504656666376587944248131632L-2869i
r15:
5.99999999999999999999999999999999999999999999682191255265L0
4.922274257721006804327508158757962070795161104520353410442L-2871i
r16:
3.9999999999999999999999999999999999999999999999999999999999L0
-1.450199512569617547104205552561340213056847944215908051543L-2874i
r17:
3.000000000000000000000000000000000000000000000256074L0
5.17181601916958516857379498520426382347254803102022654095L-2875i
r18:
8.000000000000000000000000000000000000000000000103615448L0
-5.16649480573263021415289266233934822028131384685602715104L-2870i
r19:
2.0000000000000000000000000000000000000000000000000000000000L0
-4.3863671422429801721323455460919316a48197360000025412929904L-2877i
r20:
1.0000000000000000000000000000000000000000000000000000000000L0
4.2147127594091673255332351730736563879181933855782536744554L-2947i

Verifying roots...
|p(r1)| = 4.193098383001176199674526026261267692613659354468809786439L-31
|p(r2)| = 1.197896334413083252151850733780535862482369477562027157945L-33
|p(r3)| = 2.2060094358859081861421489536218586849635659970969869285522L-32
|p(r4)| = 4.71831905228584464177582761834127942973424519919893485036L-32
|p(r5)| = 4.863607876527214564220647422757041491974728741445121202243L-35
|p(r6)| = 1.357895662302588007794306689610954386865078136883217745710L-31
|p(r7)| = 7.1495722487568807282796240594496890769142979748118075491L-31
|p(r8)| = 4.763258827515774466784629947864957497818539683166087547L-33
|p(r9)| = 1.30039062559715555556941947934032102981086898772179122802669L-36
|p(r10)| = 8.949497756751408816472620087245594672064504356749723574L-32
|p(r11)| = 3.16198789474379847883137179539836809982759000036953152655L-32
|p(r12)| = 1.38862194095550816565672495361184964857885720533195521645L-33
|p(r13)| = 3.376313649419352947632035190365951481146214543628336535L-34
|p(r14)| = 2.37964137644586862739442110275932859403416108649494676354L-36
|p(r15)| = 1.76321452623343126193104805833368516727998335158131228226L-37
|p(r16)| = 3.085627670908054708417933412028394902739970865267297439605L-38
|p(r17)| = 1.77662759900885427590041954606378444995526815548466981332307L-34
|p(r18)| = 3.67341984631946846242024301678819517743183329869412773540715L-39
|p(r19)| = 5.1299156812713547085884175552664809519965145933760624614L-2930

The 'L' characters in the numbers above are due to CLN's way of representing “Long-Float” objects. It has the same function as the 'e' character often encountered in numbers written in scientific notation.

4.2 Program tests

What follows are some tests of generating some basic amoebas from their XML files. For reference, these tests were done using a computer with a Dual-Core AMD Opteron 165 1.8 GHz processor, running 64-bit GNU/Linux (gentoo). The amoeba and Newton polytope images they generated are presented in figures 1 to 5.

Test 1

This is a test of the bivariate polynomial

\[ p(z, w) = w - 2z - 1. \]

```xml
<xml version="1.0" encoding="UTF-8"/>
<amoeba name="Amoeba 1">
    <image filename="amoeba1.png" width="640" height="640"/>
    <latex filename="amoeba1.tex"/>
    <data filename="amoeba1.dat"/>
    <npolytope filename="amoeba1npt.png"/>
</amoeba>
<settings>
    <max lnZ="10"/>
    <min lnZ="-10"/>
</settings>
<polynomial>
    <!-- p(z, w) = w - 2z - 1 -->
    <term wexp="1"/>
    <term coeff="-2" zexp="1"/>
    <term coeff="-1"/>
</polynomial>
<interval type="square" from="-0.001-0.001i" to="0.001+0.001i" step="0.0001"/>
```
$\text{time amoeba -a amoebal.xml}$

Saving data in amoebal.dat
Using 2 parallel processes.
Computing #1(8): $-0.001L0-0.001L0i$ ... $0.001L0+0.001L0i$, step=1.0L-4
Computing #2(8): $-0.01L0-0.01L0i$ ... $0.01L0+0.01L0i$, step=0.001L0
Computing #3(8): $-0.1L0-0.1L0i$ ... $0.1L0+0.1L0i$, step=0.1L0
Computing #4(8): $-1.0L0-1.0L0i$ ... $1.0L0+1.0L0i$, step=0.1L0
Computing #5(8): $-3.0L0-3.0L0i$ ... $3.0L0+3.0L0i$, step=0.3L0
Computing #6(8): $-6.0L0-6.0L0i$ ... $6.0L0+6.0L0i$, step=0.6L0
Computing #7(8): $-10.0L0-10.0L0i$ ... $10.0L0+10.0L0i$, step=1.0L0
Computing #8(8): $-100.0L0-100.0L0i$ ... $100.0L0+100.0L0i$, step=10.0L0
Saved image in amoebal.png
Saved Newton Polytope in amoebalnpt.png
Saved LaTeX in amoebal.tex

real  0m0.191s
user  0m0.280s
sys   0m0.000s

Figure 1: Newton polytope and amoeba for $p(z, w)$ in test 1
Test 2

This is a test of the bivariate polynomial

\[ p(z, w) = 3z^2 + 5zw + w^3 - 1. \]

1. `<xml version="1.0" encoding="UTF-8"?>`
2. `<amoeba name="Amoeba 2">`
3. `<description>A simple amoeba.</description>`
4. `<image filename="amoeba2.png" width="640" height="480"/>
5. `<latex filename="amoeba2.tex"/>
6. `<npolytope filename="amoeba2npt.png"/>
7. `<settings>
8. `<max precision="20"/>
9. `<max error="1e-10"/>
10. `</settings>`
11. `<polynomial>
12. `<!-- p(z, w) = 3z^2 + 5zw + w^3 + 1 -->
13. `<term coeff="3" zexp="2"/>
14. `<term coeff="5" zexp="1" wexp="1"/>
15. `<term wexp="3"/>
16. `<term coeff="-1"/>
17. `</polynomial>`
18. `<interval type="square" from="-0.001-0.001i" to="0.001+0.001i" step="0.0001"/>
19. `<interval type="square" from="-0.1-0.1i" to="0.1+0.1i" step="0.01"/>
20. `<interval type="square" from="-1.0-1.0i" to="1.0+1.0i" step="0.1"/>
21. `<interval type="square" from="-3.0-3.0i" to="3.0+3.0i" step="0.3"/>
22. `<interval type="square" from="-6.0-6.0i" to="6.0+6.0i" step="0.6"/>
23. `<interval type="square" from="-10.0-10.0i" to="10.0+10.0i" step="1.0"/>
24. `<interval type="square" from="-100.0-100.0i" to="100.0+100.0i" step="10.0"/>
25. `</amoeba>`

$ time amoeba -a amoeba2.xml
Saving data in amoeba.dat
Using 2 parallel processes.
Computing #1(7): -0.001L0-0.001L0i ... 0.001L0+0.001L0i, step=1.0L-4
Computing #2(7): -0.1L0-0.1L0i ... 0.1L0+0.1L0i, step=0.01L0
Computing #3(7): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.1L0
Computing #4(7): -3.0L0-3.0L0i ... 3.0L0+3.0L0i, step=0.3L0
Computing #5(7): -6.0L0-6.0L0i ... 6.0L0+6.0L0i, step=0.6L0
Computing #6(7): -10.0L0-10.0L0i ... 10.0L0+10.0L0i, step=1.0L0
Computing #7(7): -100.0L0-100.0L0i ... 100.0L0+100.0L0i, step=10.0L0
Saved image in amoeba2.png
Saved Newton Polytope in amoeba2npt.png
Saved LaTeX in amoeba2.tex

real   0m0.937s
user   0m1.590s
sys    0m0.030s

Test 3

This is a test of the bivariate polynomial

\[ p(z, w) = 1 + z + z^2 + z^3 + z^2w^3 + 10zw + 12z^2w + 10z^2w^2. \]

1. `<xml version="1.0" encoding="UTF-8"?>`
2. `<amoeba name="Amoeba 3">`
3. `<description>A simple amoeba.</description>`
4. `<image filename="amoeba3.png" width="640" height="480"/>
5. `<latex filename="amoeba3.tex"/>`
Figure 2: Newton polytope and amoeba for $p(z, w)$ in test 2

```
<npolytope filename="amoeba3npt.png"/>
<settings>
  <max precision="50"/>
</settings>
<polynomial>
  <!-- $p(z, w) = 1 + z + z^2 + z^3 + z^2w^3 + 10zw + 12z^2w + 10z^2w^2$ -->
  <term coeff="1"/>
  <term zexp="1"/>
  <term zexp="2"/>
  <term zexp="3"/>
  <term zexp="2" wexp="3"/>
  <term coeff="10" zexp="1" wexp="1"/>
  <term coeff="12" zexp="2" wexp="1"/>
  <term coeff="10" zexp="2" wexp="2"/>
</polynomial>

<!-- circle intervals -->
<interval type="circle" minradius="0.0001" maxradius="0.001" step="0.00001" points="20"/>
<interval type="circle" minradius="0.001" maxradius="0.01" step="0.0001" points="20"/>
<interval type="circle" minradius="0.01" maxradius="0.1" step="0.001" points="20"/>
<interval type="circle" minradius="0.1" maxradius="1.0" step="0.01" points="20"/>

<!-- square intervals -->
<interval type="square" from="-0.001-0.001i" to="0.001+0.001i" step="0.0001"/>
<interval type="square" from="-0.1-0.1i" to="0.1+0.1i" step="0.01"/>
<interval type="square" from="-1.0-1.0i" to="1.0+1.0i" step="0.1"/>
<interval type="square" from="-3.0-3.0i" to="3.0+3.0i" step="0.3"/>
<interval type="square" from="-6.0-6.0i" to="6.0+6.0i" step="0.6"/>
<interval type="square" from="-10.0-10.0i" to="10.0+10.0i" step="1.0"/>
<interval type="square" from="-100.0-100.0i" to="100.0+100.0i" step="10.0"/>
</amoeba>

$ time amoeba -a amoeba3.xml
Saving data in amoeba.dat
Using 2 parallel processes.
Computing #1(13): 1.0L-4 ... 0.001L0, step=1.0L-5, points=20
Computing #2(13): 0.001L0 ... 0.01L0, step=1.0L-4, points=20
```
Computing #3(13): 0.01L0 ... 0.1L0, step=0.001L0, points=20
Computing #4(13): 0.1L0 ... 1.0L0, step=0.01L0, points=20
Computing #5(13): 1.0L0 ... 10.0L0, step=1.0L0, points=20
Computing #7(13): -0.001L0-0.001L0i ... 0.001L0+0.001L0i, step=1.0L-4
Computing #8(13): -0.1L0-0.1L0i ... 0.1L0+0.1L0i, step=0.01L0
Computing #9(13): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.1L0
Computing #10(13): -3.0L0-3.0L0i ... 3.0L0+3.0L0i, step=0.3L0
Computing #12(13): -10.0L0-10.0L0i ... 10.0L0+10.0L0i, step=1.0L0
Computing #13(13): -100.0L0-100.0L0i ... 100.0L0+100.0L0i, step=10.0L0
Saved image in amoeba3.png
Saved Newton Polytope in amoeba3npt.png
Saved LaTeX in amoeba3.tex

real 0m10.372s
user 0m20.420s
sys 0m0.010s

Figure 3: Newton polytope and amoeba for $p(z, w)$ in test 3

Test 4

This is a test of the bivariate polynomial

$$p(z, w) = 50z^3 + 83z^2w + 24zw^2 + w^3 + 392z^2 + 414zw + 50w^2 - 28z + 59w - 100.$$
\begin{equation}
p(z,w) = 50z^3 + 83z^2w + 24zw^2 + w^3 + 414z^2 + 50w^2 - 28z + 59w - 100
\end{equation}

\begin{verbatim}
$ time amoeba -a amoeba4.xml
Saving data in amoeba4.dat
Using 2 parallel processes.
Computing #1(15): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.5L0
Computing #2(15): 1.0L-4 ... 0.001L0, step=1.0L-6, points=20
Computing #3(15): 0.001L0 ... 0.01L0, step=0.001L0, points=20
Computing #4(15): 0.01L0 ... 0.1L0, step=0.001L0, points=20
Computing #5(15): 0.1L0 ... 1.0L0, step=0.01L0, points=20
Computing #6(15): 1.0L0 ... 10.0L0, step=0.1L0, points=20
Computing #7(15): 10.0L0 ... 100.0L0, step=1.0L0, points=20
Computing #8(15): 100.0L0 ... 1000.0L0, step=1.0L0, points=20
Computing #9(15): -0.001L0-0.001L0i ... 0.001L0+0.001L0i, step=1.0L-4
Computing #10(15): -0.1L0-0.1L0i ... 0.1L0+0.1L0i, step=0.1L0
Computing #11(15): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.1L0
Computing #12(15): -3.0L0-3.0L0i ... 3.0L0+3.0L0i, step=0.3L0
Computing #13(15): -6.0L0-6.0L0i ... 6.0L0+6.0L0i, step=0.6L0
Computing #14(15): -10.0L0-10.0L0i ... 10.0L0+10.0L0i, step=1.0L0
Computing #15(15): -100.0L0-100.0L0i ... 100.0L0+100.0L0i, step=10.0L0
Saved image in amoeba4.png
Saved Newton Polytope in amoeba4npt.png
Saved LaTeX in amoeba4.tex

real  1m37.054s
user  2m52.130s
sys   0m0.200s
\end{verbatim}
Test 5

This is a test of the bivariate polynomial

\[ p(z, w) = z + \cos(w). \]
$\text{time amoeba -a amoeba5.xml}$

Saving data in amoeba5.dat

Using 2 parallel processes.

Computing #1(13): 1.0L-4 ... 0.001L0, step=1.0L-5, points=3
Computing #2(13): 0.001L0 ... 0.01L0, step=1.0L-4, points=3
Computing #3(13): 0.01L0 ... 1.0L0, step=0.001L0, points=3
Computing #4(13): 1.0L0 ... 10.0L0, step=0.1L0, points=3
Computing #6(13): 10.0L0 ... 100.0L0, step=1.0L0, points=3
Computing #7(13): -0.001L0-0.001L0i ... 0.001L0+0.001L0i, step=1.0L-4
Computing #8(13): -0.1L0-0.1L0i ... 0.1L0+0.1L0i, step=0.01L0
Computing #9(13): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.1L0
Computing #10(13): -3.0L0-3.0L0i ... 3.0L0+3.0L0i, step=0.3L0
Computing #11(13): -6.0L0-6.0L0i ... 6.0L0+6.0L0i, step=0.6L0
Computing #12(13): -10.0L0-10.0L0i ... 10.0L0+10.0L0i, step=1.0L0
Computing #13(13): -100.0L0-100.0L0i ... 100.0L0+100.0L0i, step=10.0L0

Saved image in amoeba5.png
Saved Newton Polytope in amoeba5npt.png
Saved LaTeX in amoeba5.tex

real 1m24.249s
user 2m37.750s
sys 0m0.090s

Test 6: Power series

This is a test of the power polynomial

\[ p(z, w) = \sum_{j=0}^{4} z(-1)^j \frac{w^{2j+1}}{(2j+1)!}, \]

which is actually \( z \sin(w) \) over 5 terms.

\[ \text{xml version}="1.0" encoding="UTF-8"? \]
\[ \text{amoeba name}="Amoeba 6" \]
\[ \text{description}=\text{This sum is actually } z\sin(w) \text{ over 55 terms.} \]
\[ \text{latex filename}=\text{amoeba6.tex} \]
\[ \text{data filename}=\text{amoeba6.dat} \]
\[ \text{npolytope filename}=\text{amoeba6npt.png} \]
\[ \text{settings} \]
\[ \text{max precision}=20 \]
\[ \text{max error}=1e-8 \]
\[ \text{powerseries from}=0 \text{ to}=4 \]
\[ !-- z\sin(w) := \text{sum}((-1)^j w^{2j+1}/(2j+1)!, n=0..4) --> \]
\[ !-- first term: (-1)^j z --> \]
\[ !-- second term: [((2j+1)!)/(-1)^j w^{2j+1}] --> \]
Figure 5: Newton polytope and amoeba for \( p(z, w) \) in test 5

$\text{time amoeba -a amoeba6.xml}$

Saving data in amoeba6.dat

Using 2 parallel processes.
Computing #1(7): -0.001L0-0.001L0i ... 0.001L0+0.001L0i, step=1.0L-4
Computing #2(7): -0.1L0-0.1L0i ... 0.1L0+0.1L0i, step=0.1L0
Computing #3(7): -1.0L0-1.0L0i ... 1.0L0+1.0L0i, step=0.1L0
Computing #4(7): -3.0L0-3.0L0i ... 3.0L0+3.0L0i, step=0.3L0
Computing #5(7): -6.0L0-6.0L0i ... 6.0L0+6.0L0i, step=0.6L0
Computing #6(7): -10.0L0-10.0L0i ... 10.0L0+10.0L0i, step=1.0L0
Computing #7(7): -100.0L0-100.0L0i ... 100.0L0+100.0L0i, step=10.0L0
Saved image in amoeba6.png
Saved Newton Polytope in amoeba6npt.png
Saved LaTeX in amoeba6.tex

real 0m9.732s
5 Discussion

5.1 The result

The program does what it was intended to do and it meets the requirements. We have focused on the core of the amoeba creation, so we have not implement any graphical user interface (GUI). Maybe the natural extension of the program would be to also implement a GUI. But as one gain more confidence with editing the XML-configuration file by hand, one comes to insight that a GUI is not really necessary.

Further extensions and/or improvements might be to extend the list of functions available for polynomials and power series in the XML-configuration file. One other improvement could be more image manipulating features, but with some basic knowledge in using an image-manipulating program, one can always manipulate the image afterwards.

And by modifying the intervals into 3D-structures (i.e box and a ball), maybe not too much work would have to be done to extend the limitation of the two-variate polynomial into a three-variate polynomial. But of course, the time complexity of the computations would increase enourmosly.

5.2 About the work

We started our work by implementing the amoeba-algorithm in Maple. And then worked on the Maple-code until it evolved into a state which was good enough that
we could move it over to C++. During this time we also searched for open source libraries that could and would fit our needs, such as image creation, arbitrary precision numbers, and an appropriate root-finding algorithm.

Why we chose C++ over other programming languages was mainly because of its efficiency, amount of available open source libraries, and fast execution. And we both know the language fairly well, and of course, speed was one major demand from our own side. The platform of development has been GNU/Linux. And the collaboration was done using an excellent version control system, Subversion [SVN]. For documentation we chose \LaTeX\, and divided the document in different files so we could more easily work on the text independently.

References

[CLN] Class Library for Numbers, \url{http://www.ginac.de/CLN/}


[GD] The GD library, \url{http://www.libgd.org/}


[ratfun] Polynomials And Rational Functions, \url{http://calcrpnpy.sourceforge.net/ratfunManual.html}
A Description of command line options

Our program uses Argp and therefore understands UNIX-style argument vectors. What follows is a description of each command line option.

-a, --amoeba=AMOEBA.xml
Plots an amoeba as defined in an XML file.
This option must be included if one wants to generate an amoeba from the polynomial specified in an XML file.

-h, --height=HEIGHT
Height of the image to generate from the list.
This is optional and should be used together with the --plot or -p option. It defines the height (in pixels) of the image that is to be constructed from a file containing $x$- and $y$-values.

-o, --output=IMAGEFILE
Name of the png file to plot – used in conjunction with --plot or -p.

-p, --plot=LISTFILE
Plots an image from a predefined list of $x$- and $y$-values.
This is optional but useful if one has a file containing $x$- and $y$-values of floating point numbers.

-s, --skip
Skip computing the amoeba, use data file from previous run – used in conjunction with the -a option.
This is optional but useful if an amoeba has already been computed but one wants to change the dimensions of the image or add/remove the coordinate axes. All one has to do then is to edit the image dimensions or axes option in the XML file and reconstruct the amoeba using the -a or --amoeba option together with this option. What will happen is that the program will use the data file from the previous run to construct the amoeba instead of going through the tedious process of polynomial root extraction all over again.

-v, --verbose
Verbose mode. Dump values read from XML file etc.

-w, --width=WIDTH
Width of the image to generate from the list.
This is optional and should be used together with the --plot or -p option. It defines the width (in pixels) of the image that is to be constructed from a file containing x- and y-values.

-?, --help
  Give this help list.

--usage
  Give a short usage message.

-V, --version
  Print program version.

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

B The structure of the XML config file

The configuration of the amoeba is defined as follows, with its default values given. Note: XML tags are case sensitive.

<amoeba name="Amoeba">...
  Mandatory
  Has one optional argument, name. This will be the title of the \LaTeX{} document.
</amoeba>

<description>...
  Optional
  The description of the amoeba. Default is “An amoeba”. This text is also inserted in the \LaTeX{} document.
</description>

<image filename="amoeba.png" width="640" height="480" axes="no">
  Optional
  The file name of the generated image. Set axes="yes" to also produce coordinate axes.
</image>

<latex filename="amoeba.tex">
  Optional
  The name of the generated \LaTeX{} document.
</latex>

<data filename="amoeba.dat">
  Optional
  The name of the data file used when computing the amoeba.
</data>

<npolytope filename="npolytope.png">
  Optional
  The file name of the Newton polytope image.
</npolytope>

<settings>...
  Optional
  Must be defined before polynomial. It should contain the XML tags <max .../> and <min .../>
<max precision="50"/>
Optional
The precision used when computing the amoeba. Valid range is 15 to 1024.

<max error="1e-30"/>
Optional
The maximum error used when finding roots. Valid range is 1e−30 to 1e−2.

<max processes="2"/>
Optional
The maximum number of parallel processes used when computing. Valid range is 1 to 1024.

<max lnZ="20.0"/>
Optional
The maximum lnZ value used when creating the amoeba image.

<min lnZ="-5.0"/>
Optional
The minimum lnZ value used when creating the amoeba image.

<max lnW="10.0"/>
Optional
The maximum lnW value used when creating the amoeba image.

<min lnW="-10.0"/>
Optional
The minimum lnW value used when creating the amoeba image.

<polynomial>...</polynomial>
Mandatory
Within this tag you define your polynomial using <term .../>.

<powerseries from="0" to="0">...</powerseries>
Mandatory
Within this tag you define your power series using <term .../>. Valid range of the from– and to–values are 1 to 1024 and that the to–value must be larger than the from–value.

<term coeff="a+bi" zexp="0" wexp="0"/>
Part of a polynomial.
A term in the polynomial on the form \((a + bi)z^{\text{zexp}}w^{\text{wexp}}\).

<term coeff="a+bi" function="exp|sin|cos" var="z|w" terms="10"/>
Part of a polynomial.
A term in the polynomial on the form \((a + bi)\text{exp}|\sin|\cos(z|w), and terms are the number of terms used when doing the Taylor expansion of the desired function.

<term coeff="a+bi" cexp="mj+n" var="z|w" vexp="mj+n"/>
Part of a power series.
Defines a term in a power series with coeff as a complex number, var is the variable and cexp and vexp are valid j-expressions\(^1\).

\[<\text{term jcoeff="mj+n" cexp="mj+n" var="z|w" vexp="mj+n"/>}\]

Part of a power series.
Defines a term in a power series where var is the variable and jcoeff, cexp and vexp are valid j-expressions.

\[<\text{term jcoeff="mj+n" function="fac|abs" cexp="mj+n" var="z|w" vexp="mj+n"/>}\]

Part of a power series.
As above, but here the function fac (factorial) or abs (absolute value) is applied to jcoeff.

\[<\text{interval type="square" from="a+bi" to="c+di" step="0.0"/>}\]

Mandatory
Defines an interval of type square, where from is the lower left corner and to is the upper right corner, and step defines how small or large each step should be.

\[<\text{interval type="circle" minradius="0.0" maxradius="0.0" step="0.0" points="0"/>}\]

Mandatory
Defines an interval of type circle (actually a disc approximated by a set of circles), with minimum radius minradius and maximum radius maxradius, step defines how small or large each step should be between the radii of each circle, and points is the amount of points on the circumference of each circle.

\[C \text{ Taylor tests}\]

What follows is a series of approximations for the amoeba of \(f(z, w) = z + \cos(w)\).
The pictures (figures 7 to 12) are the amoebas of \(p(z, w) = z + T_k(\cos(w))\) for \(k = 2 \ldots 19\), where \(T_k(\cos(w))\) is the Taylor approximation for \(\cos(w)\) at the origin, having \(k\) terms.

---

\(^1\)An expression on the form \(mj + n\) where \(m\) and \(n\) are integers, e.g. \(2j + 1\), \(3\), \(j - 1\), \(-j\), \(5j\) or \(-4\).
Figure 7: $z + T_k(\cos(w))$,  $k = 2 \ldots 4$

Figure 8: $z + T_k(\cos(w))$,  $k = 5 \ldots 7$

Figure 9: $z + T_k(\cos(w))$,  $k = 8 \ldots 10$
Figure 10: $z + T_k(\cos(w))$, $k = 11 \ldots 13$

Figure 11: $z + T_k(\cos(w))$, $k = 14 \ldots 16$

Figure 12: $z + T_k(\cos(w))$, $k = 17 \ldots 19$


D Source code

amoeba.h

1 //
2 // $Id: amoeba.h 1883 2007-08-12 00:19:14Z magnus $
3 //
4 // Project: Amoeba Program
5 // Authors: Magnus Leksell <nfk03ml@student.hig.se>,
6 // Wojciech Komorowski <nmd04wki@student.hig.se>
7 //
8 #ifndef AMOEBA_H_
9 #define AMOEBA_H_
10 #endif AMOEBA_H_
11 #ifndef AMOEBA_H_
12 #define AMOEBA_H_
13 #endif AMOEBA_H_
14
15 #include <cln/univpoly_complex.h>
16 #include <list>
17 #include <stdexcept>
18 #include <semaphore.h>
19 #include "rootfinders/CPSolver.h"
20 #include "interval.h"
21 #include "point.h"
22 #include "cpolynomial.h"
23
24 using namespace cln;
25
26 // The type of the values in the list of (ln|z|, ln|w|) points that make up
27 // the amoeba
28 typedef pair<double, double> AmoebaListDataType;
29
30 // The list itself
31 class AmoebaList : public list<AmoebaListDataType> {
32 public:
33     void push_back(double x, double y);
34 }
35
36 class Amoeba {
37     int imageWidth_; // width of the image in pixels
38     int imageHeight_; // height of the image in pixels
39     int precision_; // number of bits in the mantissa part of each arbitrary
40     // precision number that is going to be used
41     cl_R maxError_;
42
43     // the limits on the values of (ln|z|, ln|w|) that make up the amoeba
44     double minLnW_, maxLnW_, minLnZ_, maxLnZ_;
45
46     // this stores the list of (ln|z|, ln|w|) points so that it can be reused
47     // in consecutive runs of the program
48     string dataFileName_;
49
50     // the polynomial root finder
51     CPSolver solver_;
52
53 public:
54     Amoeba();
55     ~Amoeba();
56
57     // sets the image dimensions and limits on the values of ln|z| and ln|w|
58     void setDimensions(int width, int height, double minLnW, double maxLnW,
59                        double minLnZ, double maxLnZ);
60
61     // sets the amount of mantissa bits for arbitrary precision floating point
62     // variables
63     void setPrecision(int precision);
64
65     void setMaxError(const cl_R& maxError);
// sets a name for the data file
void setDataFileName(const std::string& dataFileName);

/*
This method computes the amoeba of the bivariate polynomial \( p(z, w) \)
by replacing one of the variables by a constant to get an univariate
polynomial, say \( p_z(w) \). The roots of \( p_z(w) = 0 \) are then located, and
the pair \((\ln|z|, \ln|\text{root}|)\) is inserted into a list, for all the roots
of \( p_z(w) \). The same procedure is applied by replacing the other variable
by a constant to get \( p_w(z) = 0 \) and adding \((\ln|\text{root}|, \ln|w|)\) into the list.
The set of constant values which replace one of the variables in \( p(z, w) \)
is taken from an AbstractInterval representing a Square or a Circle.
If the AbstractInterval represents a Square, the interval is a rectangular
region in the complex plane where the set of constants are evenly
distributed. Otherwise it is a disk centered at the origin, modelled as a set of circles centered at the origin, with increasing radius,
and the set of constants is evenly distributed along the circumference of
each circle.
*/
void compute(const CPolynomial& p, const AbstractInterval* interval,
    sem_t* mutex);

// maps the list of \((\ln|z|, \ln|w|)\) points to pixels
Points mapToImage(bool wantsAxes = false) throw(std::runtime_error);
// void mapToImage(const char* fileName) throw(std::runtime_error);

private:

// these methods substitute one of the variables in the bivariate polynomial
// \( p(z, w) \) to create an univariate polynomial \( p_z(w) \) or \( p_w(z) \)
cl_UP_N createP_Z(const CPolynomial& p, const cl_N& w);
cl_UP_N createP_W(const CPolynomial& p, const cl_N& z);

#endif /* AMOEBA_H_ */
#include "square.h"
#include "circle.h"

using namespace std;
using namespace cln;

void AmoebaList::push_back(double x, double y) {
    list<AmoebaListDataType>::push_back(AmoebaListDataType(x, y));
}

Amoeba::Amoeba() {
}

Amoeba::~Amoeba() {
}

void Amoeba::setDimensions(int width, int height, double minLnW, double maxLnW,
                         double minLnZ, double maxLnZ) {
    imageWidth_ = width;
    imageHeight_ = height;
    minLnW_ = minLnW;
    maxLnW_ = maxLnW;
    minLnZ_ = minLnZ;
    maxLnZ_ = maxLnZ;
}

void Amoeba::setPrecision(int precision) {
    precision_ = precision;
}

void Amoeba::setMaxError(const cl_R& maxError) {
    maxError_ = maxError;
}

void Amoeba::setDataFileName(const string& dataFileName) {
    dataFileName_ = dataFileName;
}

void Amoeba::compute(const CPolynomial& p, const AbstractInterval* interval,
                     sem_t* mutex) {
    cl_inhibit_floating_point_underflow = cl_false;

    // The list of (ln|z|, ln|w|) points that make up the amoeba
    AmoebaList amoebaList;

    // Determine the type of interval
    if (typeid(*interval) == typeid(Square)) {
        // Iterate through the rectangular region in the complex plane
        for (Number y = imagpart(interval->from()); y <= imagpart(interval->to()); y = y + interval->step()) {
            for (Number x = realpart(interval->from()); x <= realpart(interval->to()); x = x + interval->step()) {
                // This variable holds the constant which will substitute one of the
                // variables in p(z, w) = 0.
                cl_N subs = complex(x, y);

                // subs cannot be 0 since ln(0) is undefined
                if (zerop(subs))
                    continue;

                // Now substitute each of the variables in p(z, w) with the
                // subs constant and thus create two univariate polynomials
                cl_UP_N p_z = createP_Z(p, subs);
                cl_UP_N p_w = createP_W(p, subs);
            }
        }
    } else {
        // Iterate through each of the points in the grid
        for (Number z = imagpart(interval->from()); z <= imagpart(interval->to()); z = z + interval->step()) {
            for (Number w = realpart(interval->from()); w <= realpart(interval->to()); w = w + interval->step()) {
                // This variable holds the constant which will substitute one of the
                // variables in p(z, w) = 0.
                cl_N subs = complex(z, w);

                // subs cannot be 0 since ln(0) is undefined
                if (zerop(subs))
                    continue;

                // Now substitute each of the variables in p(z, w) with the
                // subs constant and thus create two univariate polynomials
                cl_UP_N p_z = createP_Z(p, subs);
                cl_UP_N p_w = createP_W(p, subs);
            }
        }
    }
}
```cpp
// Take the natural logarithm of the absolute value of the substituted variable
double lnAbsSubs = double_approx(ln(abs(subs)));

// Now find the roots of p_z(w) = 0 and p_w(z) = 0
list<cl_N> pz_roots = solver_.getRoots(p_z, float_format(precision_), maxError_);
list<cl_N> pw_roots = solver_.getRoots(p_w, float_format(precision_), maxError_);

// For each root of p_w(z) = 0...
for (; pzIter != pz_roots.end(); ++pzIter) {
    // If it is 0, ignore it, since ln|0| is undefined
    if (zerop(*pzIter))
        continue;

    // Get the natural logarithm of the absolute value of the root
    double lnAbsRoot = double_approx(ln(abs(*pzIter)));

    // If p(z, w) = 0 but ln|z| or ln|w| falls outside the user defined limits, then those values should be ignored...
    if (lnAbsSubs < minLnW_ || lnAbsSubs > maxLnW_)
        continue;
    if (lnAbsRoot < minLnZ_ || lnAbsRoot > maxLnZ_)
        continue;

    // Otherwise, we have a new (ln|z|, ln|w|) point of the amoeba!
    amoebaList.push_back(lnAbsRoot, lnAbsSubs);
}

// Now do the same for each root of p_z(w) = 0
for (; pwIter != pw_roots.end(); ++pwIter) {
    if (zerop(*pwIter))
        continue;

    double lnAbsRoot = double_approx(ln(abs(*pwIter)));

    // If p(z, w) = 0 but ln|z| or ln|w| falls outside the user defined limits, then those values should be ignored...
    if (lnAbsSubs < minLnZ_ || lnAbsSubs > maxLnZ_)
        continue;
    if (lnAbsRoot < minLnW_ || lnAbsRoot > maxLnW_)
        continue;

    // Otherwise, we have a new (ln|z|, ln|w|) point of the amoeba!
    amoebaList.push_back(lnAbsSubs, lnAbsRoot);
}

// The interval is of type Circle
else {
    // Iterate through all the circles
    for (Number radius = interval->minRadius();
        radius <= interval->maxRadius();
        radius = radius + interval->step()) {

        // Get the amount of substitute variables on each circle
        int points = interval->points();

        cl_F Pi = pi(float_format(precision_));
        cl_F random = random_F(Pi); // 0 <= random < Pi

        // For each substitute variable
        for (int i = 0; i < points; i++) {

            // Create the substitute!
            // the +random is used so that the points get more evenly
```
// distributed around the circumference of each circle
cl_N subs = radius * cis((2 * i * Pi) / points) + random;

// Create the univariate polynomials by substituting one of
// the variables in p(z, w)
cl_UP_N p_z = createP_Z(p, subs);
cl_UP_N p_w = createP_W(p, subs);

// Find the roots of p_z(w) = 0 and p_w(z) = 0
list<cl_N> pz_roots = solver_.getRoots(p_z, float_format(precision_), maxError_);
list<cl_N> pw_roots = solver_.getRoots(p_w, float_format(precision_), maxError_);
list<cl_N>::const_iterator pzIter = pz_roots.begin();
list<cl_N>::const_iterator pwIter = pw_roots.begin();

// Take the natural logarithm of the absolute value of the
// substituted variable
double lnAbsSubs = double_approx(ln(abs(subs)));

// For each root of p_w(z) = 0...
for (; pzIter != pz_roots.end(); ++pzIter) {
  // If it is 0, ignore it, since ln(0) is undefined
  if (zerop(*pzIter))
    continue;

  // Get the natural logarithm of the absolute value of the root
  double lnAbsRoot = double_approx(ln(abs(*pzIter)));

  // If p(z, w) = 0 but |z| or |w| falls outside the
  // user defined limits, then those values should be ignored...
  if (lnAbsSubs < minLnZ_ || lnAbsSubs > maxLnZ_)
    continue;
  if (lnAbsRoot < minLnW_ || lnAbsRoot > maxLnW_)
    continue;

  // Otherwise we have a new (ln|z|, ln|w|) point of the amoeba!
  amoebaList.push_back(lnAbsRoot, lnAbsSubs);
}

// Now do the same for each root of p_z(w) = 0
for (; pwIter != pw_roots.end(); ++pwIter) {
  // If it is 0, ignore it, since ln(0) is undefined
  if (zerop(*pwIter))
    continue;

  // Get the natural logarithm of the absolute value of the root
  double lnAbsRoot = double_approx(ln(abs(*pwIter)));

  // If p(z, w) = 0 but |z| or |w| falls outside the
  // user defined limits, then those values should be ignored...
  if (lnAbsSubs < minLnZ_ || lnAbsSubs > maxLnZ_)
    continue;
  if (lnAbsRoot < minLnW_ || lnAbsRoot > maxLnW_)
    continue;

  // Otherwise we have a new (ln|z|, ln|w|) point of the amoeba!
  amoebaList.push_back(lnAbsSubs, lnAbsRoot);
}

// mutex lock
cout << "Waiting..." << endl;
if (sem_wait(mutex) != 0) {
  perror("Mutex wait error");
  exit(1);
}

// cout << " writing... ";
// Save to file
ofstream os(dataFileName_.c_str(), ios_base::out | ios_base::binary | ios_base::app);
AmeobaList::const_iterator iter = amoebaList.begin();
for (; iter != amoebaList.end(); ++iter) {
double x = (*iter).first;
double y = (*iter).second;

os.write((char *)&x, sizeof(double));

os.write((char *)&y, sizeof(double));
}
os.close();

// cout << "done." << endl;

// mutex unlock
if (sem_post(mutex) != 0) {
    perror("Mutex post error");
    exit(1);
}
}

// This method maps the list of \(\ln|z|, \ln|w|\) points to an image with
// integer pixel coordinates

Points Amoeba::mapToImage(bool wantsAxes) throw(std::runtime_error) {
    Points points;
    AmoebaList amoebaList;

    // these are used to determine the minimum and maximum values of
    // all the \(\ln|z|\)'s and \(\ln|w|\)'s in the list, so that the image could
    // be scaled appropriately
    double minX, maxX, minY, maxY;
    minX = maxX = minY = maxY = 0;

    // Read points from data file
    ifstream is(dataFileName_.c_str(), ios_base::binary);

    if (!is)
        throw std::runtime_error(string("Could not open " + dataFileName_.c_str()));

    while (is.good()) {
        double x;
        double y;
        is.read((char *)&x, sizeof(double));
        is.read((char *)&y, sizeof(double));

        if (x < minX)
            minX = x;
        if (x > maxX)
            maxX = x;
        if (y < minY)
            minY = y;
        if (y > maxY)
            maxY = y;

        amoebaList.push_back(x, y);
    }
    is.close();

    double dx = (maxX - minX)/(double)imageWidth_;
    double dy = (minY - maxY)/(double)imageHeight_;

    /* minX + (pixel_x)*dx = ln|z|
     * minY + (pixel_y)*dy = ln|w|
     * pixel_x = (ln|z| - minX)/dx
     * pixel_y = (ln|w| - minY)/dy
     */

    //**** DEBUG: make coord axes ****/
    //**** shall we make this an option? ****/
    if (wantsAxes) {
        double origo_x = -minX/dx;
        double origo_y = -maxY/dy;
    }
```cpp
int foox = (int)origo_x;
int barx = (int)origo_y;

for (int i = 0; i < imageHeight_; i++) {
    Key key(foox, i);
    points[key] = DEFAULT_COLOR;
}

for (int i = 0; i < imageWidth_; i++) {
    Key key(i, barx);
    points[key] = DEFAULT_COLOR;
}

/******************************************************************/

// MOVE and adjust the points to fit an image structure.
while (!amoebaList.empty()) {
    AmoebaListDataType xy = amoebaList.front();
    int Xpix = (int)((xy.first - minX)/dx);
    int Ypix = (int)((xy.second - maxY)/dy);
    Key key(Xpix, Ypix);
    points[key] = DEFAULT_COLOR;
    amoebaList.pop_front();
}

return points;
}

// Private method createP_Z
// Creates a polynomial p(z) from the given polynomial p(z, w)
// by substituting w in p(z, w) with a constant
cl_UP_N Amoeba::createP_Z(const CPolynomial& p, const cl_N& w) {
    cl_univpoly_complex_ring R = find_univpoly_ring(cl_C_ring);
    cl_UP_N p_z = R->zero();
    p_z.finalize();

    CPolynomial::const_iterator it;
    for (it = p.begin(); it != p.end(); ++it) {
        int z_exp = (*it).first.first;
        int w_exp = (*it).first.second;
        cl_UP_N tmp = R->create(z_exp);
        tmp.set_coeff(z_exp, expt(w, cl_I(w_exp))*(*it).second);
        tmp.finalize();
        p_z = p_z + tmp;
    }

    return p_z;
}

// Private method createP_W
// Creates a polynomial p(w) from the given polynomial p(z, w)
// by substituting z in p(z, w) with a constant
cl_UP_N Amoeba::createP_W(const CPolynomial& p, const cl_N& z) {
    cl_univpoly_complex_ring R = find_univpoly_ring(cl_C_ring);
    cl_UP_N p_w = R->zero();
    p_w.finalize();

    CPolynomial::const_iterator it;
    for (it = p.begin(); it != p.end(); ++it) {
        int z_exp = (*it).first.first;
        int w_exp = (*it).first.second;
        cl_UP_N tmp = R->create(w_exp);
        tmp.set_coeff(w_exp, expt(z, cl_I(z_exp))*(*it).second);
        tmp.finalize();
        p_w = p_w + tmp;
    }

    return p_w;
}
```

circle.h

1  //
2  // $Id: circle.h 1684 2007-07-23 14:15:04Z magnus $
3  //
4  // Project: Amoeba Program
5  // Authors: Magnus Leksell <nfk03ml@student.hig.se>,
6  // Wojciech Komorowski <nmd04wki@student.hig.se>
7  //
8  ifndef CIRCLE_H_
9  define CIRCLE_H_
10
11  #include <queue>
12  #include <cln/complex_io.h>
13  #include "interval.h"
14
15  using namespace std;
16
17  class Circle : public AbstractInterval {
18
19  public:
20    Circle(const Number& minRadius, const Number& maxRadius, const Number& step, int points) {  
21      minRadius_ = minRadius;
22      maxRadius_ = maxRadius;
23      step_ = step;
24      points_ = points;
25    }
26
27    ostream& print(ostream& os) const {  
28      os << minRadius_ << " ... " << maxRadius_ << ", step=" << step_ << ", points=" << points_;
29      return os;
30    }
31  };
32
33  #endif /* CIRCLE_H_ */

color.h

1  //
2  // $Id: color.h 1757 2007-07-27 16:12:28Z cortex $
3  //
4  // Project: Amoeba Program
5  // Authors: Magnus Leksell <nfk03ml@student.hig.se>,
6  // Wojciech Komorowski <nmd04wki@student.hig.se>
7  //
8  ifndef COLOR_H_
9  define COLOR_H_
10
11  typedef int Color;
12  const Color DEFAULT_COLOR = 0;  // Amoebas will get plotted with this color
13
14  #endif /* COLOR_H_ */

complex.h

1  //
3  //
```cpp
#ifndef COMPLEX_H_
#define COMPLEX_H_

#include <cln/complex.h>
// #include "number.h"

using namespace cln;

typedef cln::cl_N Complex;

/*
 * struct ComplexConverter {
 * static const Number& getRe(const Complex& c) {
 * return the<Number>(realpart(c));
 * }
 * static const Number& getIm(const Complex& c) {
 * return the<Number>(imagpart(c));
 * }
 * }
 */
#endif /* COMPLEX_H_ */

conffile.h

#ifndef CONFFILE_H_
#define CONFFILE_H_

#include <string>
#include <list>
#include <cln/lfloat.h>
#include <cln/complex.h>
#include <libxml++.h>
#include "environment.h"
#include "number.h"
#include "cpolynomial.h"
#include "interval.h"

// Class ConfFile

class ConfFile : public Environment, protected xmlpp::SaxParser {
  bool onDescription_;  
  bool onPowerSeries_;  
  bool firstTime_;      
  bool verbose_;        
  std::map<int, cln::cl_N> zPoly_;  
  std::map<int, cln::cl_N> wPoly_;  

  // Private methods
  cln::lfloat createLF(const string& x);  
  cln::cl_N createComplexLF(const string& x);  
  void createTaylor(const cln::cl_N coeff, const string& fun,
```
const string& var, unsigned int terms);
void taylorexpSin(const cln::cl_N& coeff, const string& var, unsigned int terms);
void taylorexpCos(const cln::cl_N& coeff, const string& var, unsigned int terms);
void taylorexpExp(const cln::cl_N& coeff, const string& var, unsigned int terms);
bool validate(const string& exp);
int eval(const string& exp, unsigned int j);
void createPowerSeries(const cln::cl_N& coeff, const string& var, const string& cexp, const string& vexp, const string& jcoeff, const string& fun);
void addSeriesToPolyString(const cln::cl_N& coeff, const string& var, const string& cexp, const string& vexp, const string& jcoeff, const string& fun);
void addToPolyString(const cln::cl_N& coeff, const string& var, const string& cexp, const string& vexp, const string& jcoeff, const string& fun);
void setAmoeba(const AttributeList& properties);
void setSettings(const AttributeList& properties);
void setImage(const AttributeList& properties);
void setLatex(const AttributeList& properties);
void setData(const AttributeList& properties);
void setNPolytope(const AttributeList& properties);
void setPolynomial(const AttributeList& properties);
void setPowerSeries(const AttributeList& properties);
void setTerm(const AttributeList& properties);
void setInterval(const AttributeList& properties);
void setMax(const AttributeList& properties);
void setMin(const AttributeList& properties);

protected:
// Overrides xmlpp:
virtual void on_start_document();
virtual void on_end_document();
virtual void on_start_element(const Glib::ustring& name, const AttributeList& properties);
virtual void on_end_element(const Glib::ustring& name);
virtual void on_characters(const Glib::ustring& characters);
virtual void on_comment(const Glib::ustring& text);
virtual void on_warning(const Glib::ustring& text);
virtual void on_error(const Glib::ustring& text);
virtual void on_fatal_error(const Glib::ustring& text);

public:
ConfFile(bool verbose = false);
~ConfFile();
// Overrides Environment:
//const CPolynomial& getPolynomial();
void parseFile(const string& filename);

#include <iostream>

#endif /* CONFFILE_H_ */

conffile.cpp
```cpp
#include <sstream>
#include <stdexcept>
#include <cln/cln.h>
#include <sys/types.h>
#include <regex.h>
#include "conffile.h"
#include "square.h"
#include "circle.h"

using namespace cln;

// Constructor
ConfFile::ConfFile(bool verbose) {
  onDescription_ = false;
  onPowerSeries_ = false;
  firstTime_ = true;
  verbose_ = verbose;
}

// Destructor
ConfFile::~ConfFile() {
  // Deallocate intervals
  while (!intervals_.empty()) {
    AbstractInterval* i = intervals_.front();
    delete i;
    intervals_.pop_front();
  }
}

// Private method createLF
cl_LF ConfFile::createLF(const string& x) {
  ostringstream os;
  //os << x + "L+0_" << precision_;  
  os << x << "_" << precision_;
  return cl_LF(os.str().c_str());
}

// Private method createComplexLF
cl_N ConfFile::createComplexLF(const string& x) {
  // Read a complex
  istringstream is(x);
  cl_N dummy;
  is >> dummy;
  //float_format_t prec = float_format(precision_);
  // Create cl_LF of the realpart
  */
```

ostringstream os;
    os << realpart(dummy);
    cl_LF a = createLF(os.str());
    //cl_F a = cl_float(realpart(dummy), prec);

    // Create cl_LF of the imagpart
    os.str("");
    os << imagpart(dummy);
    cl_LF b = createLF(os.str());
    //cl_F b = cl_float(imagpart(dummy), prec);

    return complex(a, b);
}

// Private method createTaylor
// This method creates the Taylor expansion for some functions so that
// functions of the type f(z, w) = z + cos(w) can be approximated by
// bivariate polynomials p(z, w) = z + 1 - w^2/2! + ...

void ConfFile::createTaylor(const cl_N& coeff, const string& fun,
    const string& var, unsigned int terms) {
    if (var != "z" && var != "w")
        throw runtime_error("Unknown variable. Check spelling in XML file");

    if (fun == "sin")
        taylorexpSin(coeff, var, terms);
    else if (fun == "cos")
        taylorexpCos(coeff, var, terms);
    else if (fun == "exp")
        taylorexpExp(coeff, var, terms);
    else
        throw runtime_error("Unknown function. Check spelling in XML file");
}

// Creates a Taylor expansion for the sine function
// sin z = z - z^3/3! + z^5/5! - ...

void ConfFile::taylorexpSin(const cl_N& coeff, const string& var,
    unsigned int terms) {
    bool negative = false;
    bool z_var = (var == "z");

    for (unsigned int i = 0; i < terms; i++) {
        unsigned int exponent = 2*i+1;
        cl_I fact = factorial(exponent);
        if (z_var) {
            cl_N tmp = poly_[Key(exponent, 0)];
            if (negative)
                tmp = tmp - coeff/fact;
            else
                tmp = tmp + coeff/fact;
            poly_[Key(exponent, 0)] = tmp;
        } else {
            cl_N tmp = poly_[Key(0, exponent)];
            if (negative)
                tmp = tmp - coeff/fact;
            else
                tmp = tmp + coeff/fact;
            poly_[Key(0, exponent)] = tmp;
        }
        negative = !negative;
    }
}
/ * Creates a Taylor expansion for the cosine function
  * \( \cos z = 1 - \frac{z^2}{2!} + \frac{z^4}{4!} - ... \)
  *
  */
void ConfFile::taylorexpCos(const cl_N& coeff, const string& var,
    unsigned int terms) {
    bool negative = false;
    bool z_var = (var == "z");
    for (unsigned int i = 0; i < terms; i++) {
        unsigned int exponent = 2*i;
        cl_I fact = factorial(exponent);
        if (z_var) {
            cl_N tmp = poly_[Key(exponent, 0)];
            if (negative)
                tmp = tmp - coeff/fact;
            else
                tmp = tmp + coeff/fact;
            poly_[Key(exponent, 0)] = tmp;
        } else {
            cl_N tmp = poly_[Key(0, exponent)];
            if (negative)
                tmp = tmp - coeff/fact;
            else
                tmp = tmp + coeff/fact;
            poly_[Key(0, exponent)] = tmp;
        }
        negative = !negative;
    }
}

// * Creates a Taylor expansion for the exponential function
// \( \exp z = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \)
//
void ConfFile::taylorexpExp(const cl_N& coeff, const string& var,
    unsigned int terms) {
    bool z_var = (var == "z");
    for (unsigned int i = 0; i < terms; i++) {
        cl_I fact = factorial(i);
        if (z_var) {
            cl_N tmp = poly_[Key(i, 0)];
            tmp = tmp + coeff/fact;
            poly_[Key(i, 0)] = tmp;
        } else {
            cl_N tmp = poly_[Key(0, i)];
            tmp = tmp + coeff/fact;
            poly_[Key(0, i)] = tmp;
        }
    }
}

// Private method validate
//
bool ConfFile::validate(const string& exp) {
    if (exp == "j" || exp == "-j")
        return true;
    // This accepts patterns like (n, m integer):
    // 0, j, -j, j+n, j-n, mj, -mj, mj+n, mj-n, -mj+n, -mj-n, m, -m
    // Or shortly, [-]m[+]n[-]m
string expr = "\^[0-]\{0,1\}\{0-9\}([0-9]*\{0,1\}0,1\}\$";
expr += \{0,1\}[j]\{0,1\}0,1\}\$";
regex_t preg;

// Compile the expression
if (regcomp(&preg, expr.c_str(), REG_NOSUB) != 0) {
// This should never happen!
cerr \ "regcomp error!" \ endl;
return false;
}

// Validate the exponent
int status = regexec(&preg, exp.c_str(), (size_t)0, NULL, 0);
regfree(&preg);
return status == 0;

// Private method eval
// Evaluates the exponent string to a integer.
// Precondition: That the string exp is a valid
// expression according to the validate method.
int ConfFile::eval(const string& exp, unsigned int j) {
int m = 1;
int n = 0;
int xj = 1;
string str = exp;
int jindex = exp.find_first_of('j');

if (str == "j")
xj = j;
else if (str == "-j")
xj = -j;
else if (jindex == 1 && exp[0] == '-') {
// "-j..."
str.replace(0, 2, 1, ' ');
// Remove the "-j" substring
xj = -j;
istringstream is(str);
is >> n;
// Read one integer
}
else if (jindex == 0) {
// "i..."
str.replace(jindex, 1, 1, ' ');
// Remove the 'j' character
xj = j;
istringstream is(str);
is >> n;
// Read one integer
}
else if (jindex > 0) {
// "...j..."
str.replace(jindex, 1, 1, ' ');
// Remove the 'j' character
xj = j;
istringstream is(str);
is >> m;
// Get the first integer
if (is.good())
// If there is a second integer:
is >> n;
// get it
}
else {
// No 'j' in exp
istringstream is(str);
is >> m;
// Get the first integer
if (is.good())
// If there is a second integer:
is >> n;
// get it
}

int value = m * xj + n;
if (verbose_) {
    cout << " j=" << j << " , eval(" << exp << ") : " << m << "*" << xj
    << (n<0)?"":"") << n << "=" << value << endl;
}
return value;
}

// Private method createPowerSeries

void ConfFile::createPowerSeries(const cl_N& coeff, const string& var,
const string& cexp, const string& vexp,
const string& jcoeff, const string& fun) {
    if (var != "z" && var != "w")
        throw runtime_error("Unknown variable. Check spelling in XML file");
    bool z_var = (var == "z");
    for (unsigned int j = powerSeriesRange_.first
         ; j <= powerSeriesRange_.second; j++) {
        int zexp = 0;
        int wexp = 0;
        if (z_var)
            zexp = eval(vexp, j);
        else
            wexp = eval(vexp, j);
        // Validate exponents
        if (zexp < 0)
            throw out_of_range("z-exponent below 0");
        if (wexp < 0)
            throw out_of_range("w-exponent below 0");
        int cj = eval(cexp, j);
        cl_I jcoeffValue = 1;
        if (jcoeff != "") {
            jcoeffValue = eval(jcoeff, j);
            if (fun != "") {
                if (fun == "fac")
                    jcoeffValue = factorial(cl_I_to_uint(jcoeffValue));
                else if (fun == "abs")
                    jcoeffValue = abs(jcoeffValue);
                else
                    throw runtime_error("Unknown function. Check spelling in XML file");
            }
        }
        cl_N tmp;
        //if (z_var)
        tmp = zPoly_[zexp];
        else
        tmp = wPoly_[wexp];
        tmp = poly_[Key(zexp, wexp)];
        if (tmp == complex(0, 0))
            tmp = complex(1, 0);
        // DEBUG
}
/**cout << "tmp=\" << tmp << ", "
**<< "coeff=\" << coeff << ", "
**<< "jcoeffValue=\" << jcoeffValue << ", "
**<< "cj=\" << cj << endl;*/

// multiply with current coeff
tmp = tmp * expt(jcoeffValue, cj) * expt(coeff, cj);

// if (z_var)
zPoly_[zexp] = tmp;
else
wPoly_[wexp] = tmp;/

poly_[Key(zexp, wexp)] = tmp;

if (verbose_)
{ cout << " added: (\" << tmp
<< ")^\" << zexp << "w^" << wexp << endl;
}
}
}

// Private method addSeriesToPolyString

void ConfFile::addSeriesToPolyString(const cl_N& coeff, const string& var,
const string& cexp, const string& vexp,
const string& jcoeff, const string& fun) {

ostringstream os;

// TeXify the power series
if (firstTime_)
{ os << "\sum_{j=\" << powerSeriesRange_.first << "}^{\" <<
powerSeriesRange_.second << "}\";
firstTime_ = false;
}

if (realpart(coeff) != 1 || imagpart(coeff) != 0)
{ os << "(\" << coeff << ")^\" << cexp << "\";
}

if (jcoeff != "")
{ if (fun != "")
{ if (fun == "fac")
{ os << "\[\" << jcoeff << "]!^\" << cexp << "\";
}
else
{ os << "\[\" << fun << "\" << jcoeff << "]!^\" << cexp << "\";
}
else
{ os << "(" << jcoeff << ")^\" << cexp << "\";
}

os << var << "^\" << vexp << "\";

polyString_ += os.str();
}
}

// Private method addToPolyString

void ConfFile::addToPolyString(const cl_N& coeff, unsigned int zexp,
unsigned int wexp) {

ostringstream os;

if (!firstTime_ && realpart(coeff) >= 0)
{ os << "+";
}
if (zexp == 0 && wexp == 0)
    os << coeff;
else if (realpart(coeff) != 1 || imagpart(coeff) != 0)
    if (imagpart(coeff) == 0)
        os << coeff;
    else
        os << "(" << coeff << ")";
if (zexp == 1)
    os << "z";
else if (zexp != 0)
    os << "z^" << zexp;
if (wexp == 1)
    os << "w";
else if (wexp != 0)
    os << "w^" << wexp;
firstTime_ = false;
polyString_ += os.str();
} 

// Private method addToPolyString
//
void ConfFile::addToPolyString(const cl_N& coeff, const string& fun, 
    const string& var) {
    ostringstream os;
    if (!firstTime_ && realpart(coeff) >= 0)
        os << "+";
    if (realpart(coeff) != 1 || imagpart(coeff) != 0)
        os << coeff;
    // TeXify the function
    os << "/\" << fun << "(" << var << ")";
    firstTime_ = false;
polyString_ += os.str();
} 

// Private method setAmoeba
//
void ConfFile::setAmoeba(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        if (iter->name == "name")
            amoebaName_ = iter->value;
        else
            cout << "Warning: Unknown XML entity: " << iter->name << " endl;
    }
} 

// Private method setSettings
//
void ConfFile::setSettings(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        if (iter->name == "precision")
            precision_ = atoi(iter->value.c_str());
else if (iter->name == "iterations")
iterations_ = atoi(iter->value.c_str());
else if (iter->name == "rooterror")
    // TODO: rootError_ = ?
}
*/

void ConfFile::setImage(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        //cout << " Attribute " << iter->name << " = " << iter->value << endl;
        if (iter->name == "filename")
            imageFileName_ = iter->value;
        else if (iter->name == "width")
            imageWidth_ = atoi(iter->value.c_str());
        else if (iter->name == "height")
            imageHeight_ = atoi(iter->value.c_str());
        else if (iter->name == "axes")
            wantsAxes_ = iter->value == "yes";
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
}

void ConfFile::setLatex(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        //cout << " Attribute " << iter->name << " = " << iter->value << endl;
        if (iter->name == "filename")
            latexFileName_ = iter->value;
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
}

void ConfFile::setData(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        //cout << " Attribute " << iter->name << " = " << iter->value << endl;
        if (iter->name == "filename")
            dataFileName_ = iter->value;
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
}

void ConfFile::setNPolytope(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        //cout << " Attribute " << iter->name << " = " << iter->value << endl;
        if (iter->name == "filename")
            newtonPolytopeFileName_ = iter->value;
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
}
cout << "Warning: Unknown XML entity: " << iter->name << endl;
}

// Private method setPolynomial

void ConfFile::setPolynomial(const AttributeList& properties) {
if (verbose_)
    cout << "Parsing polynomial..." << endl;

AttributeList::const_iterator iter = properties.begin();
for (; iter != properties.end(); ++iter) {
    //cout << " Attribute " << iter->name << " = " << iter->value << endl;
}
}

// Private method setPowerSeries

void ConfFile::setPowerSeries(const AttributeList& properties) {
unsigned int from = 0;
unsigned int to = 0;
cl_N coeff = complex(1, 0);
string var = "*";

if (verbose_)
    cout << "Parsing power series, ";
for (; iter != properties.end(); ++iter) {
    if (iter->name == "from")
        from = atoi(iter->value.c_str());
    else if (iter->name == "to")
        to = atoi(iter->value.c_str());
    else
        cout << "Warning: Unknown XML entity: " << iter->name << endl;
}

if (verbose_)
    cout << "j=" << from << "." << to << endl;

// Verify some values
if (from >= to)
    throw runtime_error("to-value not larger than from-value");
if (to > 1024)
    throw out_of_range("to-value larger than 1024");

onPowerSeries_ = true;
powerSeriesRange_.first = from;
powerSeriesRange_.second = to;
}

// Private method setTerm

void ConfFile::setTerm(const AttributeList& properties) {
cl_N coeff = complex(1, 0);
unsigned int zexp = 0;
unsigned int wexp = 0;
bool gotFunction = false;
string fun = "*";
string var = "*";
string cexp = "j"; // coeff exponent
string vexp = "j"; // var exponent
unsigned int terms = 10; // default
string jcoeff = "";

AttributeList::const_iterator iter = properties.begin();
for (; iter != properties.end(); ++iter) {
    if (iter->name == "coeff") {
        istringstream is(iter->value.c_str());
        is >> coeff;
    } else if (iter->name == "jcoeff") {
        jcoeff = iter->value;
    } else if (iter->name == "zexp")
        zexp = atoi(iter->value.c_str());
    else if (iter->name == "wexp")
        wexp = atoi(iter->value.c_str());
    else if (iter->name == "function") {
        fun = iter->value;
        gotFunction = true;
    } else if (iter->name == "var")
        var = iter->value;
    else if (iter->name == "terms")
        terms = atoi(iter->value.c_str());
    else if (iter->name == "cexp")
        cexp = iter->value;
    else if (iter->name == "vexp")
        vexp = iter->value;
    else
        cout << "Warning: Unknown XML entity: " << iter->name << endl;
}

// Print some info
if (verbose_)
    if (onPowerSeries_)
        cout << " adding term: (" << coeff << ")^(" << cexp << ")^\{z\}^\{w\}^\{ terms \}/^\{ vexp \}^\{ gotFunction \}
        if (jcoeff != "")
            if (gotFunction)
                cout << " adding function: (" << coeff << ")" << fun << ">
                    cout << " adding term: (" << coeff << ")z^\{ cexp \}^\{ terms \}/^\{ vexp \}^\{ gotFunction \}
        if (terms < 0 || terms > 1024)
            throw out_of_range("terms out of range 0..1024");
    if (wexp < 0 || wexp > 1024)
        throw out_of_range("w exponent out of range 0..1024");
    if (terms < 0 || terms > 1024)
        throw out_of_range("terms out of range 0..1024");
    if (!validate(cexp))
        throw runtime_error("coeff-exponent not a valid j-expression");
if (!validate(vexp))
  throw runtime_error("var-exponent not a valid j-expression");
if (jcoeff != "" && !validate(jcoeff))
  throw runtime_error("jcoeff not a valid j-expression");

// Add to the polynomial
if (onPowerSeries_) {
  addSeriesToPolyString(coeff, var, cexp, vexp, jcoeff, fun);
  createPowerSeries(coeff, var, cexp, vexp, jcoeff, fun);
} else if (gotFunction) {
  addToPolyString(coeff, fun, var);
  createTaylor(coeff, fun, var, terms);
} else {
  addToPolyString(coeff, zexp, wexp);
  poly_[Key(zexp, wexp)] = coeff;
}

// Private method setInterval

void ConfFile::setInterval(const AttributeList& properties) {
  Glib::ustring type;
  Complex from = complex(0, 0);
  Complex to = complex(0, 0);
  Number minRadius = 0.0;
  Number maxRadius = 0.0;
  Number step = 0.0;
  unsigned int points = 0;
  AttributeList::const_iterator iter = properties.begin();
  for (; iter != properties.end(); ++iter) {
    if (iter->name == "type")
      type = iter->value;
    else if (iter->name == "minradius")
      minRadius = createLF(iter->value);
    else if (iter->name == "maxradius")
      maxRadius = createLF(iter->value);
    else if (iter->name == "step")
      step = createLF(iter->value);
    else if (iter->name == "points")
      points = atoi(iter->value.c_str());
    else if (iter->name == "from") {
      from = createComplexLF(iter->value);
    } else if (iter->name == "to") {
      to = createComplexLF(iter->value);
    } else {
      cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
    if (type == "circle")
      intervals_.push_back(new Circle(minRadius, maxRadius, step, points));
    else if (type == "square")
      intervals_.push_back(new Square(from, to, step));
    else
      throw runtime_error("Unknown interval type");
  }

  // Private method setMax

void ConfFile::setMax(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        if (iter->name == "precision")
            precision_ = atoi(iter->value.c_str());
        else if (iter->name == "error")
            maxError_ = cl_R(iter->value.c_str());
        else if (iter->name == "processes")
            maxProcesses_ = atoi(iter->value.c_str());
        else if (iter->name == "lnZ")
            maxLnZ_ = atof(iter->value.c_str());
        else if (iter->name == "lnW")
            maxLnW_ = atof(iter->value.c_str());
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }

    // Validate some data
    if (precision_ < 15 || precision_ > 1024)
        throw out_of_range("precision out of range 15..1024");
    if (maxError_ < cl_R("1e-30") || maxError_ > cl_R("1e-2"))
        throw out_of_range("max error out of range 1e-30..1e-2");
    if (maxProcesses_ < 1 || maxProcesses_ > 1024)
        throw out_of_range("max processes out of range 1..1024");
}

void ConfFile::setMin(const AttributeList& properties) {
    AttributeList::const_iterator iter = properties.begin();
    for (; iter != properties.end(); ++iter) {
        if (iter->name == "lnZ")
            minLnZ_ = atof(iter->value.c_str());
        else if (iter->name == "lnW")
            minLnW_ = atof(iter->value.c_str());
        else
            cout << "Warning: Unknown XML entity: " << iter->name << endl;
    }
}

void ConfFile::on_start_document() {
    //cout << "on_start_document()" << endl;
}

void ConfFile::on_end_document() {
    //cout << "on_end_document()" << endl;
}

void ConfFile::on_start_element(const Glib::ustring& name, const AttributeList& attributes) {
    // Check node name (case sensitive) and call appropriate method.
    if (name == "amoeba")
        setAmeoba(attributes);
    else if (name == "description")
        onDescription_ = true;
    else if (name == "settings")
        setSettings(attributes);
    else if (name == "image")
        setImage(attributes);
    else if (name == "latex")
        setLatex(attributes);
    else if (name == "data")
setData(attributes);
else if (name == "npolytope")
    setNPolytope(attributes);
else if (name == "polynomial")
    setPolynomial(attributes);
else if (name == "powerseries")
    setPowerSeries(attributes);
else if (name == "term")
    setTerm(attributes);
else if (name == "interval")
    setInterval(attributes);
else if (name == "max")
    setMax(attributes);
else if (name == "min")
    setMin(attributes);
else
    cerr << "Warning: Unknown XML entity: " << name << endl;
}

void ConfFile::on_end_element(const Glib::ustring& name) {
    //cout << "on_end_element()" << endl;
}

void ConfFile::on_characters(const Glib::ustring& text) {
    //cout << "on_characters(): " << text << endl;
    if (onDescription_)
        description_ = text;
    onDescription_ = false;
}

void ConfFile::on_comment(const Glib::ustring& text) {
    //cout << "on_comment(): " << text << endl;
}

void ConfFile::on_warning(const Glib::ustring& text) {
    //cout << "on_warning(): " << text << endl;
}

void ConfFile::on_error(const Glib::ustring& text) {
    //cout << "on_error(): " << text << endl;
}

void ConfFile::on_fatal_error(const Glib::ustring& text) {
    //cout << "on_fatal_error(): " << text << endl;
}

/*
 * const CPolynomial& ConfFile::getPolynomial() {
 *     if (onPowerSeries_)
 *         // Construct polynomial from the two series
 *         map<int, cl_N>::const_iterator wIter = wPoly_.begin();
 *         for (; wIter != wPoly_.end(); ++wIter) {
 *             map<int, cl_N>::const_iterator zIter = zPoly_.begin();
 *             for (; zIter != zPoly_.end(); ++zIter) {
 *                 int zexp = zIter->first;
 *                 int wexp = wIter->first;
 *                 cl_N prod = zIter->second * wIter->second;
 *                 poly_[Key(zexp, wexp)] = prod;
 *                 if (verbose_)
 *                     cout << "Added: " << prod << "z^" << zexp << "w^" << wexp << endl;
 *             }
 *         }
 *     return poly_;
 * }
 */

const CPolynomial& ConfFile::getPolynomial() {
    if (onPowerSeries_)
        // Construct polynomial from the two series
        map<int, cl_N>::const_iterator wIter = wPoly_.begin();
        for (; wIter != wPoly_.end(); ++wIter) {
            map<int, cl_N>::const_iterator zIter = zPoly_.begin();
            for (; zIter != zPoly_.end(); ++zIter) {
                int zexp = zIter->first;
                int wexp = wIter->first;
                cl_N prod = zIter->second * wIter->second;
                poly_[Key(zexp, wexp)] = prod;
                if (verbose_)
                    cout << "Added: " << prod << "z^" << zexp << "w^" << wexp << endl;
            }
        }
    return poly_;
}
```cpp
return poly_;}

/* Does not work?!
struct lessKey {
  bool operator()(const Key& a, const Key& b) const {
    return (a.first < b.first && a.second < b.second);
  }
};
*/

typedef std::map<Key, cln::cl_N> CPolynomial;

#endif /* CPOLYNOMIAL_H_ */

evironment.h

```
public:
    Environment();
    virtual ~Environment();
    virtual const CPolynomial& getPolynomial();
    virtual const string& getPolynomialString() const;
    virtual const list<AbstractInterval*>& getIntervals() const;
    virtual int getImageWidth() const;
    virtual int getImageHeight() const;
    virtual const string& getAmoebaName() const;
    virtual const string& getDescription() const;
    virtual const string& getImageFileName() const;
    virtual const string& getLatexFileName() const;
    virtual const string& getDataFileName() const;
    virtual const string& getNewtonPolytopeFileName() const;
    virtual unsigned int getPrecision() const;
    virtual const Number& getMaxError() const;
    virtual unsigned int getMaxProcesses() const;
    virtual double getMinLnW() const;
    virtual double getMaxLnW() const;
    virtual double getMinLnZ() const;
    virtual double getMaxLnZ() const;
    virtual const UIntPair& getPowerSeriesRange() const;
    virtual bool wantsAxes() const;
    virtual void parseFile(const string& filename) = 0;
friend ostream& operator << (ostream& os, const Environment& env);
};

#endif /* ENVIRONMENT_H_ */

environment.cpp

#include "environment.h"

public:
    Environment();
    virtual ~Environment();
    virtual const CPolynomial& getPolynomial();
    virtual const string& getPolynomialString() const;
    virtual const list<AbstractInterval*>& getIntervals() const;
    virtual int getImageWidth() const;
    virtual int getImageHeight() const;
    virtual const string& getAmoebaName() const;
    virtual const string& getDescription() const;
    virtual const string& getImageFileName() const;
    virtual const string& getLatexFileName() const;
    virtual const string& getDataFileName() const;
    virtual const string& getNewtonPolytopeFileName() const;
    virtual unsigned int getPrecision() const;
    virtual const Number& getMaxError() const;
    virtual unsigned int getMaxProcesses() const;
    virtual double getMinLnW() const;
    virtual double getMaxLnW() const;
    virtual double getMinLnZ() const;
    virtual double getMaxLnZ() const;
    virtual const UIntPair& getPowerSeriesRange() const;
    virtual bool wantsAxes() const;
    virtual void parseFile(const string& filename) = 0;
friend ostream& operator << (ostream& os, const Environment& env);
};
// Set some default values
polyString_ = "p(z,w)=";
imageWidth_ = 640;
imageHeight_ = 480;
amoebaName_ = "Amoeba";
description_ = "An amoeba";
imageFileName_ = "amoeba.png";
latexFileName_ = "amoeba.tex";
dataFileName_ = "amoeba.dat";
newtonPolytopeFileName_ = "npolytope.png";
precision_ = 50;
maxError_ = Number("1.0e-30");
maxProcesses_ = 2;
minLnW_ = -5.0;
maxLnW_ = 20.0;
minLnZ_ = -10.0;
maxLnZ_ = 10.0;
powerSeriesRange_.first = 0; // from
powerSeriesRange_.second = 0; // to
wantsAxes_ = false;
}

// Destructor
// Environment::~Environment() {
const CPolynomial& Environment::getPolynomial() const {
    return poly_;}
}

const string& Environment::getPolynomialString() const {
    return polyString_;}
}

const list<AbstractInterval*>& Environment::getIntervals() const {
    return intervals_;}
}

int Environment::getImageWidth() const {
    return imageWidth_;}
}

int Environment::getImageHeight() const {
    return imageHeight_;}
}

const string& Environment::getDescription() const {
    return description_;}
}

const string& Environment::getAmoebaName() const {
    return amoebaName_;}
}

const string& Environment::getImageFileName() const {
    return imageFileName_;}
}

const string& Environment::getLatexFileName() const {
    return latexFileName_;}
}

const string& Environment::getDataFileName() const {
    return dataFileName_;}
const string& Environment::getNewtonPolytopeFileName() const {
    return newtonPolytopeFileName_;}

unsigned int Environment::getPrecision() const {
    return precision_;}

const Number& Environment::getMaxError() const {
    return maxError_;}

unsigned int Environment::getMaxProcesses() const {
    return maxProcesses_;}

double Environment::getMinLnZ() const {
    return minLnZ_;}

double Environment::getMaxLnZ() const {
    return maxLnZ_;}

double Environment::getMinLnW() const {
    return minLnW_;}

double Environment::getMaxLnW() const {
    return maxLnW_;}

const UIntPair& Environment::getPowerSeriesRange() const {
    return powerSeriesRange_;}

bool Environment::wantsAxes() const {
    return wantsAxes_;}

ostream& operator << (ostream& os, const Environment& env) {
    os << "Amoeba name: " << env.amoebaName_ << endl
    << "Image file name: " << env.imageFileName_ << endl
    << "Image width: " << env.imageWidth_ << endl
    << "Image height: " << env.imageHeight_ << endl
    << "Image coord axes: " << (env.wantsAxes_?"yes":"no") << endl
    << "Description: " << env.description_ << endl
    << "LaTeX file name: " << env.latexFileName_ << endl
    << "Data file name: " << env.dataFileName_ << endl
    << "Newton polytope file name: " << env.newtonPolytopeFileName_ << endl
    << "Precision: " << env.precision_ << endl
    << "Max error: " << env.maxError_ << endl
    << "Processes: " << env.maxProcesses_ << endl
    << "Min ln w: " << env.minLnW_ << endl
    << "Max ln w: " << env.maxLnW_ << endl
    << "Min ln z: " << env.minLnZ_ << endl
    << "Max ln z: " << env.maxLnZ_ << endl
    << "Polynomial: " << env.polyString_ << endl;}

void Environment::parseFile(const string& filename) {
    // You must override this method
}
image.h

#ifndef IMAGE_H_
#define IMAGE_H_

#include "point.h"

class ImageProducer {
public:
    static void write(const char *filename, const Points& points, int width, int height)
    throw(std::runtime_error, std::out_of_range);
};
#endif /* IMAGE_H_ */

image.cpp

#include <gd.h>
#include <stdexcept>
#include "image.h"

using namespace std;

void ImageProducer::write(const char *filename, const Points& points, int width, int height) throw(runtime_error, out_of_range) {
    // Validate image size
    if (width < 1)
        throw out_of_range("Image width < 1");
    if (height < 1)
        throw out_of_range("Image height < 1");

gdImagePtr im = gdImageCreate(width, height);
int white = gdImageColorAllocate(im, 255, 255, 255); // Background color white
int black = gdImageColorAllocate(im, 0, 0, 0);

Points::const_iterator iter = points.begin();
for (; iter != points.end(); ++iter) {
    Key key = (*iter).first;
    int x = key.first;
    int y = key.second;
    if (x >= 0 && x < width && y >= 0 && y < height) {
        //cout << "setPixel (" << x << "," << y << ")" << endl;
        gdImageSetPixel(im, x, y, black);
    }
}
}
FILE* pngout = fopen(filename, "wb");
if (pngout == NULL)
    throw runtime_error("Could not save data to file!");
gdImagePng(im, pngout);
gdImageDestroy(im);
latex.h

/* INTERVAL_H_ */

#include "environment.h"

class LatexProducer {
public:
  static void write(const Environment& env) throw(runtime_error);
};

#endif /* LATEX_H_ */

latex.cpp

#include <iostream>
#include <fstream>
#include <stdexcept>
#include "latex.h"

using namespace std;

void LatexProducer::write(const Environment& env) throw(runtime_error) {
  string fileName = env.getLatexFileName();
  extern const char* argp_program_version; // quick and dirty solution!
  if (fileName == "") // Empty string -- no file name given:
    return; // silently skip this.
  ofstream os(fileName.c_str());
  if (!os)
    throw runtime_error("Could not create LaTeX file " + fileName);
  // Produce info about the amoeba
  os << "% Created by " << argp_program_version << endl
    << "% http://amoeba.sajberspejs.com" << endl
    << "% http://amoeba.sajberspejs.com" << endl
    << "\documentclass[a4paper]{article}" << endl
    << "\usepackage[english]{babel}" << endl
    << "\usepackage[T1]{fontenc}" << endl
    << "\usepackage[utf8]{inputenc}" << endl
    << "\usepackage[L1]{fontenc}" << endl
    << "\usepackage[a4]{article}" << endl

  return os;
\usepackage{amsmath} \\
\usepackage{subfigure} \\
//\usepackage[pdftex]{graphicx} \\
\usepackage{graphicx} \\
\title{env.getAmoebaName() } \\
\author{} \\
\date{} \\
\begin{document} \\
\maketitle \\
\thispagestyle{empty} \\
\begin{figure}[] \\
\centering \\
\subfigure[$\mathcal{N}_p$]{ \\
\includegraphics[width=.20\textwidth]{env.getNewtonPolytopeFileName() } } \\
\hfill \\
\subfigure[$\mathcal{A}_p$]{ \\
\includegraphics[width=.75\textwidth]{env.getImageFileName()} } \\
\caption{Newton polytope and amoeba for $p(z,w)$} \\
// env.getPolynomialString()$}$ \\
\end{figure} \\
\begin{verbatim} \\
\end{verbatim} \\
\paragraph{Description} env.getDescription() \\
\end{document} \\
// TODO: \\
// Precision/settings used. \\
// ... \\
/*
const list<AbstractInterval*> intervals = env.getIntervals(); 
list<AbstractInterval*>::const_iterator iter = intervals.begin(); 
int count = 1; 
for (; iter != intervals.end(); ++iter) { 
    os << "Interval " << count++ << "(" << intervals.size() << ": " 
        << **iter << \\
        \\
        os << "\vfill" <<endl 
        << "\noindent Created by " << argp_program_version << "\\\n        " << endl 
        << "\texttt{http://amoeba.sajberspejs.com}" << endl 
        << "\end{document}" << endl; 
    os.close(); 
} 
cout << "Saved LaTeX in " << fileName << endl; */ 

listplotter.h
```cpp
int height) throw(std::runtime_error);
private:
  static void plotList(std::istream& is, const char* outfile, int width,
   int height) throw(std::runtime_error);
};

#endif /* LISTPLOTTER_H */

listplotter.cpp

// $Id: listplotter.cpp 1943 2007-09-02 14:48:08Z magnus $
// Project: Amoeba Program
// Authors: Magnus Leksell <nfk03mlg@student.hig.se>,
// Wojciech Komorowski <nmd04wki@student.hig.se>

#include <map>
#include <list>
#include <string>
#include <iostream>
#include <fstream>
#include <stdexcept>
#include "listplotter.h"
#include "point.h"
#include "color.h"
#include "image.h"

using namespace std;

// Method plotList

void ListPlotter::plotList(const char* infile, const char* outfile, int width,
   int height) throw(runtime_error) {
  // Treat input from stdin as special case
  string str = infile;
  if (str == "-") {
    plotList(cin, outfile, width, height);
    return;
  }

  ifstream ifile(infile);
  if (!ifile)
    throw runtime_error("Could not open input file: " + str);

  cout << "Reading list of x, y values from " << infile << endl;
  try {
    plotList(ifile, outfile, width, height);
  } catch (const exception& e) {
    ifile.close();
    throw runtime_error(e.what());
  }

  ifile.close();
}

// Private method plotList

void ListPlotter::plotList(istream& is, const char* outfile, int width,
   int height) throw(runtime_error) {
  list<pair<double, double>> values;
  double minX, maxX, minY, maxY;
```

minX = maxX = minY = maxY = 0;

double xpos, ypos;

while (!is.eof()) {
    is >> xpos;
    is >> ypos;
    if (xpos < minX) minX = xpos;
    if (xpos > maxX) maxX = xpos;
    if (ypos < minY) minY = ypos;
    if (ypos > maxY) maxY = ypos;
    values.push_back(pair<double, double>(xpos, ypos));
}

double dx = (maxX - minX) / (double)width;
double dy = (minY - maxY) / (double)height;

int intx, inty;
map<Key, Color> points;
list<pair<double, double>>::const_iterator iter;
for (iter = values.begin(); iter != values.end(); iter++) {
    xpos = (*iter).first;
    ypos = (*iter).second;
    intx = (int)((xpos - minX) / dx);
    inty = (int)((ypos - maxY) / dy);
    Key key(intx, inty);
    points[key] = DEFAULT_COLOR;
}

ImageProducer::write(outfile, points, width, height);

npolytope.h

#include <gd.h>
#include "cpolynomial.h"

class NewtonPolytope {
private:
    static double angle(gdPoint p1, gdPoint p2);
    static bool leftTurn(gdPoint p1, gdPoint p2, gdPoint p3);
public:
    static void write(const char* filename,
                      const CPolynomial& p) throw(std::runtime_error);
};

npolytope.cpp

#include <gd.h>
#include "cpolynomial.h"
```cpp
#include <iostream>
#include <cmath>
#include <stdexcept>
#include <deque>
#include "gd.h"
#include "npolytope.h"
#include "point.h"

using namespace std;

void NewtonPolytope::write(const char* filename, const CPolynomial& p) throw(runtime_error) {
    if (filename == "") { // Empty string -- no file name given:
        return; // silently skip this.
    }

    int width;
    int height;
    gdImagePtr im;
    gdPoint points[p.size()];

    int index = 0;
    int maxX = 0, maxY = 0;
    for (CPolynomial::const_iterator iter = p.begin(); iter != p.end(); ++iter) {
        Key key = (*iter).first;
        unsigned int zexp = key.first;
        unsigned int wexp = key.second;
        points[index].x = zexp;
        points[index].y = wexp;
        if (zexp > maxX) maxX = zexp;
        if (wexp > maxY) maxY = wexp;
        ++index;
    }

    // if we can't draw a polytope because of too few // points, then silently skip it
    if (p.size() < 2) return;

    // find the pivot
    int pIndex = 0; // pivot index
    for (int i = 0; i < p.size(); i++) {
        if (points[i].y < points[pIndex].y) pIndex = i;
        else if (points[i].y == points[pIndex].y && points[i].x < points[pIndex].x) pIndex = i;
    }

    // place the pivot at the beginning
    if (pIndex != 0) {
        gdPoint tmp = points[0];
        points[0] = points[pIndex];
        points[pIndex] = tmp;
        pIndex = 0;
    }

    // sort the rest by increasing angle relative to the pivot
    int tmpIndex;
    double tmpAngle;
    gdPoint tmpPoint;
    gdPoint pivot = points[0];
    int j;
    for (int i = 1; i < p.size(); i++) {
        tmpIndex = i;
        tmpAngle = std::atan2(pivot.y - points[i].y, pivot.x - points[i].x);
        for (j = i; j > pIndex; j--)
            tmpIndex = j;
        tmpPoint = points[j];
        points[j] = points[i];
        points[i] = tmpPoint;
        tmpIndex = i;
        tmpAngle = std::atan2(pivot.y - points[i].y, pivot.x - points[i].x);
    }
}
``
tmpPoint = points[i];
tmpAngle = angle(pivot, points[i]);
for (j = i; j < p.size(); j++) {
    if (angle(pivot, points[j]) <= tmpAngle) {
        tmpPoint = points[j];
        tmpAngle = angle(pivot, tmpPoint);
        tmpIndex = j;
    }
}
points[tmpIndex] = points[i];
points[i] = tmpPoint;

// if several points have the same angle relative to the pivot,
// delete them except the one with the furthest distance from
// the pivot
deque<gdPoint> ptsQue;
for (int i = 0; i < p.size(); i++) {
    ptsQue.push_back(points[i]);
}

gdPoint p0 = ptsQue[0];
gdPoint p1, p2;
for (int i = 1; i < ptsQue.size()-1; i++) {
    p1 = ptsQue[i];
    p2 = ptsQue[i+1];
    if (angle(p0, p1) == angle(p0, p2)) {
        if ((p0.x-p1.x)*(p0.x-p1.x)+(p0.y-p1.y)*(p0.y-p1.y) <
            (p0.x-p2.x)*(p0.x-p2.x)+(p0.y-p2.y)*(p0.y-p2.y)) {
            i--;
            ptsQue.erase(ptsQue.begin() + (i+1));
        } else {
            i--;
            ptsQue.erase(ptsQue.begin() + (i+2));
        }
    }
    else {
        ptsQue.push_back(points[i]);
    }
}

// construct the convex hull
deque<gdPoint> hullQue;
hullQue.push_back(ptsQue[0]);
hullQue.push_back(ptsQue[1]);
gdPoint p1, p2;
int i = 2;
while (i < ptsQue.size()) {
    p1 = hullQue[hullQue.size()-2];
    p2 = hullQue[hullQue.size()-1];
    if (leftTurn(p1, p2, ptsQue[i])) {
        hullQue.push_back(ptsQue[i]);
        i++;
    } else {
        hullQue.pop_back();
    }
}
i = hullQue.size();
int hullpts = i;
gdPoint hull[1];
for (i--; i > -1; i--) {
    hull[1] = hullQue.back();
    hullQue.pop_back();
}
// setup the image
int scale = 35;
int radius = 4;
width = scale*maxX + radius/2 + 3;
height = scale*maxY + radius/2 + 3;
im = gdImageCreate(width, height);
int white = gdImageColorAllocate(im, 255, 255, 255); // Background color white
int black = gdImageColorAllocate(im, 0, 0, 0);
int grey = gdImageColorAllocate(im, 153, 153, 153);
int blue = gdImageColorAllocate(im, 0, 0, 255);

// map all points to the image
for (int i = 0; i < p.size(); i++) {
    int zexp = points[i].x;
    int wexp = points[i].y;
    points[i].x = (zexp * scale) + radius/2;
    points[i].y = height - (wexp * scale) - radius/2 - 1;
}
for (int i = 0; i < hullpts; i++) {
    int zexp = hull[i].x;
    int wexp = hull[i].y;
    hull[i].x = (zexp * scale) + radius/2;
    hull[i].y = height - (wexp * scale) - radius/2 - 1;
}

// draw points together with the hull
for (int i = 0; i < p.size(); i++) {
    int cx = points[i].x;
    int cy = points[i].y;
    gdImageFilledArc(im, cx, cy, radius, radius, 0, 360, grey, gdArc);
}
for (int i = 0; i < hullpts-1; i++) {
    gdImageLine(im, hull[i].x, hull[i].y, hull[i+1].x, hull[i+1].y, black);
}
gdImageLine(im, hull[hullpts-1].x, hull[hullpts-1].y, points[0].x, points[0].y, black);

//gdImagePolygon(im, points, p.size(), black);

FILE* pngout = fopen(filename, "wb");
if (pngout == NULL)
    throw runtime_error("Could not save image to file!");
gdImagePng(im, pngout);
gdImageDestroy(im);
cout << "Saved Newton Polytope in " << filename << endl;
}

double NewtonPolytope::angle(gdPoint p1, gdPoint p2) {
double pi = 3.14159265358979;
int Vx = p2.x - p1.x;
int Vy = p2.y - p1.y;
double norm = sqrt(Vx*Vx + Vy*Vy);
if (norm == 0.0) return norm;

double cosangle = (double)Vx / norm;
if (p1.y < p2.y) return acos(cosangle);
else if (p1.y > p2.y) return 2*pi*acos(cosangle);
else if (p1.x > p2.x) return pi;
else return 0.0;
}

bool NewtonPolytope::leftTurn(gdPoint p1, gdPoint p2, gdPoint p3) {
return ((p2.x - p1.x)*(p3.y - p1.y) - (p2.y - p1.y)*(p3.x - p1.x)) >= 0;
number.h

#include <cln/float.h>
#include <cln/float_io.h>
#include <cln/real.h>
#include <cln/real_io.h>

typedef cln::cl_R Number;

#endif /* NUMBER_H_ */

point.h

#include <map>
#include "color.h"

typedef std::pair<int, int> Key;
typedef std::map<Key, Color> Points;

#endif /* POINT_H_ */

square.h

#include <queue>
#include <cln/complex_io.h>
#include "interval.h"

using namespace std;
class Square : public AbstractInterval {
public:
    Square(const Complex& from, const Complex& to, const Number& step) {
        from_ = from;
        to_ = to;
        step_ = step;
    }

    ostream& print(ostream& os) const {
        os << from_ << " ... " << to_ << ", step=" << step_; 
        return os;
    }
};

CPSolver.h

CPSolver.cpp
```cpp
#include <iostream>
#include <cln/abort.h>
#include <cln/number.h>
#include <cln/complex.h>
#include <cln/real.h>
#include <cln/float.h>
#include <cln/sfloat.h>
#include <cln/ffloat.h>
#include <cln/dfloat.h>
#include <cln/ifloat.h>
#include <cln/rational.h>
#include <cln/integer.h>
#include <cln/univpoly_complex.h>
#include <cln/complex_io.h>
#include <cln/lfloat_io.h>
#include "CPSolver.h"

using namespace std;
using namespace cln;

// The Durand-Kerner iterative method for polynomial root finding
list<cl_N> CPSolver::getRoots(const cl_UP_N &p, float_format_t precision, const cl_R& maxError) {
    list<cl_N> roots; // where the roots will be stored
    sintL p_degree = degree(p); // degree of the polynomial

    // solve explicitly for simple polynomials
    if (p_degree < 1) { cerr << "ERROR: rootfinder called with polynomial degree < 1" << endl; cl_abort(); }
    if (p_degree == 1) {
        cl_N root = -coeff(p, 0)/coeff(p, 1);
        roots.push_back(cl_N(root));
        return roots;
    }
    else if (p_degree == 2) {
        cl_N a = coeff(p, 2);
        cl_N b = coeff(p, 1);
        cl_N c = coeff(p, 0);
        cl_N discroot = sqrt(b*b-4*a*c);
        roots.push_back(cl_N((-b+discroot)/(2*a)));
        roots.push_back(cl_N((-b-discroot)/(2*a)));
        return roots;
    }
    else {
        // set starting values by distributing them around a circle which is known to contain all the roots of the polynomial - see (1), end of Section 6.
        sintL p_degree = degree(p); // degree of the polynomial

        // consider all roots to be a cluster and take its center
        // to be the mean of all the roots, i.e. -C_(n-1)/(nC_n)
        // cl_N center = -(coeff(p, p_degree-1)/(p_degree*coeff(p, p_degree)));
        cl_RA one(1);
        cl_R radius = 1.0;
        for (int i = 1; i <= p_degree; i++) {
            cl_UP_N P = derive(p, p_degree-i);
            cl_R foo = abs(P(center)/coeff(p, p_degree));
            cl_F n = cl_float((int)p_degree, precision);
        }
    }
}
```

67
```c
cl_I fact = factorial(p_degree-i);  
n = n/fact;  
cl_N expr = n*foo;  
cl_RA blah(i);  
cl_N exponent = one/blah;  
cl_R tmp = realpart(expr, exponent));  
if (tmp > radius)  
    radius = tmp;  
}  
/*  
// distribute n numbers around the circle...  
// \{z_{(1,..,n)}\} = \{center + radius * e^{(2ikPi/n)}, k=1,...n-1\}  
cl_N r_approx[p_degree];  
cl_F Pi = pi(precision);  
for (int i = 1; i < p_degree; i++)  
    //r_approx[i] = center + radius * cis((2i*Pi)/p_degree);  
    r_approx[i] = cis((2*i*Pi)/p_degree);  
// now add the final point, so that it doesn't end up on the real axis  
//r_approx[0] = center + radius * cis(Pi/p_degree);  
r_approx[0] = cis(Pi/p_degree);  
cl_N r_prod[p_degree];  
// the leading coefficient of the polynomial must be 1 for  
// this method to converge...  
cl_univpoly_complex_ring R = find_univpoly_ring(cl_C_ring);  
cl_UP_N C_n = R->create(0);  
C_n.set_coeff(0, 1/coeff(p, p_degree));  
C_n.finalize();  
cl_UP_N P = C_n*p;  

for (unsigned int iter = 1; iter <= p_degree; iter++) {  
    for (int i = 0; i < p_degree; i++) {  
        cl_N prod = complex(1, 0);  
        for (int j = 0; j < p_degree; j++) {  
            if (j == i) continue;  
            prod = prod * (r_approx[i] - r_approx[j]);  
        }  
        cl_N tmp = r_approx[i];  
        r_approx[i] = tmp - (P(tmp)/prod);  
        iterations++;  
    }  
    // convergence = true;  
    for (int i = 0; i < p_degree; i++) {  
        cl_N prod = complex(1, 0);  
        for (int j = 0; j < p_degree; j++) {  
            if (j == i) continue;  
            prod = prod * (r_approx[i] - r_approx[j]);  
        }  
        cl_N tmp = r_approx[i];  
        r_approx[i] = tmp - (P(tmp)/prod);  
        iterations++;  
    }  
    if (iterations >= 65535) {  
        cerr << "Root finder failed after " << iterations << " iterations!" << endl;  
        for (int i = 0; i < p_degree; i++)  
        for (unsigned int iter = 1; iter <= p_degree; iter++) {  
```

68
roots.push_back(cl_N(r_approx[i]));
return roots;
}
for (int i = 0; i < p_degree; i++) {
    error = abs(p(r_approx[i]));
    if (error > maxError) {
        convergence = false;
        break;
    }
}
while (!convergence);
break;
}
for (int i = 0; i < p_degree; i++)
    roots.push_back(cl_N(r_approx[i]));
return roots;
}
inline cl_UP_N CPSolver::derive(const cl_UP_N &p, unsigned int n)
{
    cl_UP_N tmp = p;
    for (unsigned int i = 0; i < n; i++)
        tmp = deriv(tmp);
    return tmp;
}

main.cpp
static char args_doc[] = "";

static struct argp_option options[] = {
    {"plot", 'p', "LISTFILE", 0,
     "Plots an image from a predefined list of x, y values" },
    {"output", 'o', "IMAGEFILE", 0,
     "Name of the png file to plot - used in conjunction with --plot or -p" },
    {"width", 'w', "WIDTH", 0,
     "Width of the image to generate from the list" },
    {"height", 'h', "HEIGHT", 0,
     "Height of the image to generate from the list" },
    {"amoeba", 'a', "AMOEBA.xml", 0,
     "Plots an amoeba as defined in an XML file" },
    {"skip", 's', 0, 0,
     "Skip computing the amoeba, use data file from previous run - \n     used in conjunction with -a" },
    {"verbose", 'v', 0, 0, "Verbose mode. Dump values read from XML file etc."},
    { 0 }
};

struct arguments {
    bool plot, amoeba, skip, verbose;
    char* list_file;
    char* image_file;
    int width, height;
    char* amoeba_file;
};

static error_t parse_opt (int key, char *arg, struct argp_state *state) {
    struct arguments *arguments = (struct arguments*) state->input;
    switch (key) {
    case 'a':
        arguments->amoeba = true;
        arguments->amoeba_file = arg;
        break;
    case 's':
        arguments->skip = true;
        break;
    case 'v':
        arguments->verbose = true;
        break;
    case 'p':
        arguments->plot = true;
        arguments->list_file = arg;
        break;
    case 'o':
        arguments->image_file = arg;
        break;
    case 'w':
        arguments->width = atoi(arg);
        break;
    case 'h':
        arguments->height = atoi(arg);
        break;
    case ARGP_KEY_ARG:
        //if (state->arg_num >= 2)
        //argp_usage (state);
        break;
    case ARGP_KEY_END:
        if (state->argc < 2)
        argp_usage (state);
        //if (state->arg_num < 2)
        //argp_usage (state);
        break;
    default:
        return ARGP_ERR_UNKNOWN;
    }
/* Our argp parser. */
static struct argp argp = {options, parse_opt, args_doc, doc};

int main(int argc, char** argv) {
    key_t shm_key;
    int shm_id;
    sem_t* mutex;
    struct arguments arguments;

    // Create shared memory
    shm_id = shmget(IPC_PRIVATE, sizeof(sem_t), IPC_CREAT | 0600);
    if (shm_id == -1) {
        perror("Could not get shared memory");
        return 1;
    }

    // Attach to shared memory
    mutex = (sem_t*)shmat(shm_id, (void*)0, 0);
    if ((long)mutex == -1) {
        perror("Could not attach to shared memory");
        return 1;
    }

    // Create mutex
    if (sem_init(mutex, 1, 1) != 0) {
        perror("Could not create mutex");
        return 1;
    }

    arguments.plot = false;
    arguments.amoeba = false;
    arguments.skip = false;
    arguments.verbose = false;
    arguments.list_file = "-";
    arguments.image_file = "from_list.png";
    arguments.width = 640;
    arguments.height = 480;
    arguments.amoeba_file = "-";

    argp_parse(&argp, argc, argv, 0, 0, &arguments);

    if (arguments.plot)
        try {
            // Plot image and then quit
            ListPlotter::plotList(arguments.list_file, arguments.image_file,
                                  arguments.width, arguments.height);
            return 0;
        }
    catch (const exception& e) {
        cerr << "Exception caught: " << e.what() << endl;
        return 1;
    }

    if (!arguments.amoeba)
        cerr << "Could not understand arguments." << endl;
        return 1;
}

string infile = arguments.amoeba_file;
Environment* env = new ConfFile(arguments.verbose);

// Try parse the config file.
try {
    env->parseFile(infile);
    if (arguments.verbose)
        cout << *env;
    catch (const exception& e) {
        cerr << "Exception caught: " << e.what() << endl;
        return 1;
    }

    if (arguments.skip) // Skip computing the amoeba
        cout << "Skipping computing the amoeba." << endl
        << "Using data from " << env->getDataFileName() << endl;

    // Set some initial values.
    int imgWidth = env->getImageWidth();
    int imgHeight = env->getImageHeight();
    double minLnW = env->getMinLnW();
    double maxLnW = env->getMaxLnW();
    double minLnZ = env->getMinLnZ();
    double maxLnZ = env->getMaxLnZ();

    Amoeba amoeba;
    amoeba.setDataFileName(env->getDataFileName());
    amoeba.setDimensions(imgWidth, imgHeight, minLnW, maxLnW, minLnZ, maxLnZ);
    amoeba.setPrecision(env->getPrecision());
    amoeba.setMaxError(env->getMaxError());

    if (!arguments.skip) { // Calculate the amoeba
        // Remove the data file before computing, if it exists
        remove(env->getDataFileName().c_str());
        cout << "Saving data in " << env->getDataFileName() << endl
        << "Using " << env->getMaxProcesses() << endl
        << " parallel processes." << endl;

        int intervalCount = 0; // Keep track of number of intervals processed
        int processCount = 0; // Number of parallel processes currently running
        CPolynomial poly = env->getPolynomial();
        const list<AbstractInterval*> intervals = env->getIntervals();
        list<AbstractInterval*>::const_iterator iter = intervals.begin();

        while (iter != intervals.end()) { // For each interval
            pid_t child_pid;
            ++intervalCount;

            // Start a new process
            child_pid = fork();

            if (child_pid == 0) { // Child process
                cout << "Computing #" << intervalCount
                << "(" << intervals.size() << ")": " << *iter << " " << endl;
                amoeba.compute(poly, *iter, mutex);
                if (arguments.verbose)
                    cout << "...done with #" << intervalCount << endl;
                exit(0); // Finished, just exit child process
            } else if (child_pid == -1) {
                perror("Could not start new process");
                exit(1);
            }

            // Parent process
            ++processCount; // Increase number of processes running
        }
    }
}
// If maximum number of processes running,
// wait for any process to finish
if (processCount == env->getMaxProcesses()) {
    wait(NULL); // Waiting for a process to finish
    --processCount; // Decrease number of processes running
}

++iter; // Next interval

// Wait for remaining childs to finish
while (processCount > 0) {
    wait(NULL);
    --processCount;
}

// !arguments.skip

// Remove the data file when calculations is done
// if (!arguments.skip)
// remove(env->getDataFileName().c_str());

// Detach and remove shared memory
if (shmdt(mutex) == 0)
    shmctl(shm_id, IPC_RMID, NULL);

try {
    // Map the amoeba to an image structure
    // and write the image to a file
    const char* imgfile = env->getImageFileName().c_str();
    Points points = amoeba.mapToImage(env->wantsAxes());
    ImageProducer::write(imgfile, points, env->getImageWidth(),
                         env->getImageHeight());
    cout << "Saved image in " << imgfile << endl;

    // Produce the Newton polytope image
    const char* npfile = env->getNewtonPolytopeFileName().c_str();
    NewtonPolytope::write(npfile, env->getPolynomial());

    // Produce a LaTeX file.
    LatexProducer::write(*env);
}

    catch (const exception& e) {
        cerr << "Exception caught: " << e.what() << endl;
        return 1;
    }

// Done!
return 0;