Cubism

Brian Lukoff, Kyle Bradley, and Asher Walkover
with help from Roland Roeder, Bill Dunbar, and John Hubbard

August 20, 2002

1 Introduction

The Hénon maps are a generalization of the function $z \mapsto z^2 + c$ to two complex variables. The Hénon map $H : \mathbb{C}^2 \to \mathbb{C}^2$ is given by

$$H\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x^2 - ay + c \\ x \end{pmatrix}, \quad \text{with inverse} \quad H^{-1}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ \frac{1}{a}\left(y^2 + c - x\right) \end{pmatrix}, \tag{1}$$

where a and c are both complex parameters.

Two important sets are K^+ and K^-, which are the sets of points with bounded orbits under H and H^{-1}, respectively. Then the filled Julia set is $K = K^+ \cap K^-$. J^+ and J^- are the boundaries of K^+ and K^- respectively, and $J = J^+ \cap J^-$ is the Julia set.

In studying Hénon maps, two functions which are of interest are the functions G^+ and G^-, given by

$$G^+\begin{pmatrix} x \\ y \end{pmatrix} = \lim_{n\to\infty} \frac{1}{2^n} \log^+ \left\| H^n \begin{pmatrix} x \\ y \end{pmatrix} \right\| \quad \text{and} \quad G^-\begin{pmatrix} x \\ y \end{pmatrix} = \lim_{n\to\infty} \frac{1}{2^n} \log^+ \left\| H^{-n} \begin{pmatrix} x \\ y \end{pmatrix} \right\|, \tag{2}$$

where $\log^+ x = \max(0, \log x)$. (In computing G^+ and G^-, any norm can be used because they all give the same result.) G^+ and G^- measure how quickly iterates of H (the Hénon map) and H^{-1} go to infinity. We studied the surfaces

$$X_t = \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \in \mathbb{C}^2 \;\middle|\; G^+\begin{pmatrix} x \\ y \end{pmatrix} = G^-\begin{pmatrix} x \\ y \end{pmatrix} = t \right\}, \tag{3}$$

where t is a real nonnegative parameter. As $t \to 0$, X_t approaches the Julia set J. For large t, it is known that X_t is homeomorphic to a torus (and as a result has Euler characteristic 0). If x and y are both large, then $G^+\binom{x}{y} \approx \log|x|$ and $G^-\binom{x}{y} \approx \log|y|$. If t is large, then the moduli of both coordinates of all points on X_t are almost fixed, with only the arguments varying; this results in a torus. We studied what happens to X_t as t gets smaller.

2 Strategy

We studied the surfaces X_t by writing a computer program, Cubism, which, given a particular value for t, attempted to generate an approximation of X_t which accurately represents the topology of X_t.

The program is named for the method we used to compute these approximations. The general strategy is to first consider H and H^{-1} as functions from $\mathbb{R}^4$ to $\mathbb{R}^4$ (instead of $\mathbb{C}^2$ to $\mathbb{C}^2$). Then we take a region R in $\mathbb{R}^4$ (typically $[-m, m]^4$ for some $1 \leq m \leq 5$) , subdivide this region into many four-dimensional cubes, and determine which cubes contain a part of the surface X_t. Then, the collection of cubes S that are found is the desired approximation of X_t. We can analyze the topology of X_t by classifying X_t according to the Euler characteristics of each of its connected componenents. Since S should be a thickening of X_t, we can compute the Euler characteristic of X_t by computing the Euler characteristic χ of S using the formula $\chi(S) = \sum_{k=0}^{4}(-1)^k n_k$, where n_k is the number of k-dimensional objects in S.

We also generate pictures of the four three-dimensional projections of X_t, and use these pictures to get a general idea of the topology.

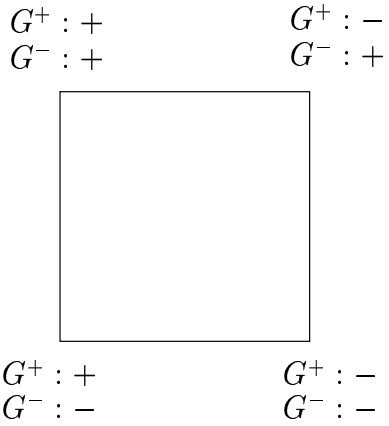
3 Finding S

To find S, we make some subdivision of R into four-dimensional cubes, and then run through each two-dimensional face that touches at least one four-dimensional cube in R (except for faces on the boundary, each two-dimensional face touches 4 four-dimensional cubes.) The intersections of a two-dimensional face with the sets $c^+ = \{G^+ = t\}$ and $c^- = \{G^- = t\}$ are each one-dimensional curves. If these curves intersect each other in the face in question, then all 4 four-dimensional cubes which touch this face are added to S. This method covers all four-dimensional cubes in R.

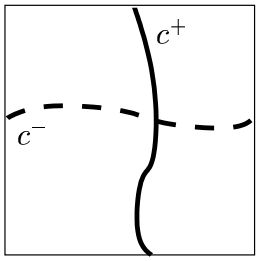
The difficulty arises in *efficiently* determining whether or not c^+ intersects c^- in a particular face. We use several algorithms to mark each face as a "yes", "maybe", or "no" (more on this later), and then mark each cube with the maximum of the markings of its two-dimensional faces (where we use the ordering "no" < "maybe" < "yes").

3.1 Vertex Decisions

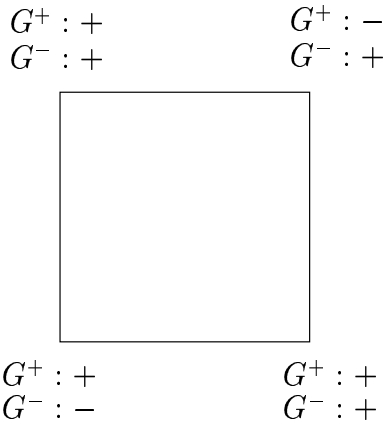
One fast method that is used to determine whether or not c^+ and c^- intersect in a particular face is by examining the signs of the functions $G^+ - t$ and $G^- - t$ at the vertices of the face. Certain combinations of signs at the vertices can ensure that c^+ will intersect c^- on a face. For example, say that the signs on a face are as below:



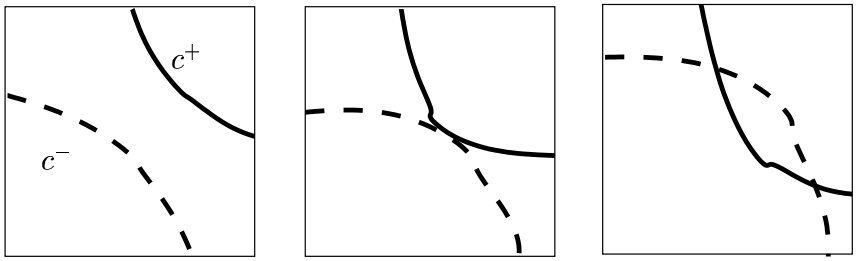
In this case, we can be sure that c^+ and c^- intersect in that face, looking something like this:



In the case above and in similar cases (when we can be sure that there is an intersection), we mark the face in question as a "yes". On the other hand, suppose that we see the combination of faces as below:



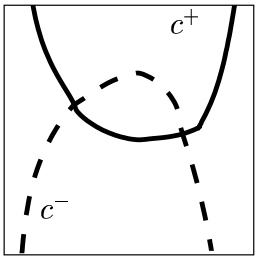
In this case, it is possible that on this particular face, c^+ and c^- may look like any of the following possibilities:



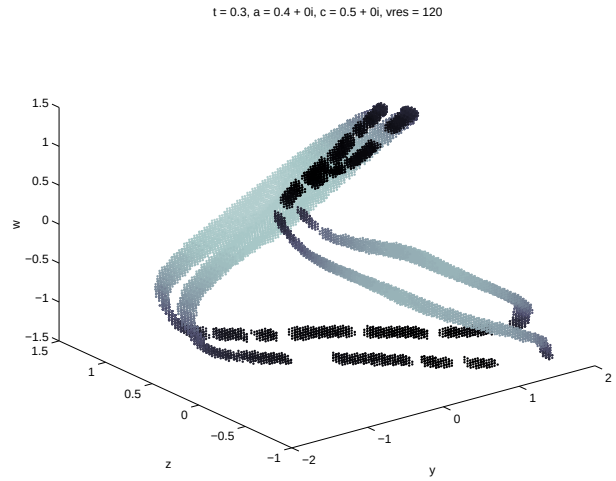
In the first possibility, there is no intersection. In the second and third possibilities, there is at least one intersection. In this case (and in others like it), we mark the face as "maybe".

Finally, if either $G^+ - t$ or $G^- - t$ is positive on all four vertices (or negative on all four vertices), then we mark the face as "no".

This method works fairly well. In particular, if this algorithm marks a face as a "yes", then c^+ and c^- definitely intersect within the face. But even if we consider S to be the set of all "yes" cubes *and* all "maybe" cubes, we still miss cubes that should be in S. This happens because either $G^+ - t$ or $G^- - t$ could change signs twice (or another even number of times), resulting in signs at the vertices that are identical to the case where the function does not change sign at all. For example, the following face would have the same signs on the vertices as a face that had no intersection of c^+ and c^-:



As a result, using only the vertex decision method resulted in approximations of X_t that had gaps in them (cubes that should have been in S but weren't). An example appears below:



If we had an infinite amount of time (and/or computational power), then we could fix the problem by making a finer subdivision of R. But Cubism takes time $O(N^4)$, if N is the number of subdivisions on a side of R, so it wasn't reasonable to simply increase N. Instead, we needed a better algorithm. In the next section, we describe how we used Newton's method to attempt to make more correct decisions about whether faces should be marked "yes", "no", or "maybe"; in section 3.3, we describe how we used a method of successive refinement to try to generate an approximation of X_t with a higher resolution.

3.2 Newton's Method

In order to mitigate this problem, we also perform an additional test on cubes that the vertex decision algorithm marks as "no". Using a two-dimensional version of Newton's method, we attempt to find a common root of the functions $G^+ - t$ and $G^- - t$, with the initial point in the center of the face. We iterate Newton's method a fixed number of times (5 seems to work well) and then try to determine whether Newton's method is converging to a root. If it is, then there is an intersection of c^+ and c^- in the face, and we mark the face as a "yes".

It is difficult to determine whether or not Newton's method is really converging to a root within the face, because although Newton's method is efficient, we are often running it on faces that do not have a root at all, and we want to separate out the behavior of Newton's method in this case (for example, converging to a root that is on another face) from the case where there is a root on the face we are looking at.

If, after the fixed number of iterations, the final iterate of Newton's method is still within the face it started in, *and* both G^+ and G^- are less than ϵ away from t, then we mark this face as a "yes". (By trial and error, we have set $\epsilon = .01$.) The second condition attempts to ensure that Newton's method is in fact converging, and the first condition attempts to ensure that Newton's method is converging to a root inside the face it started in.

3.3 Subdivision and Refinement

If we choose an initial subdivision N (so that each side of R is divided into N subdivisions), then we end up with N^4 cubes. Increasing N makes the number of cubes explode, making it infeasible to make N very large while still having Cubism finish within a reasonable amount of time. On the other hand, a larger N results in a better approximation of the surface X_t, so we want to run Cubism for as large an N as possible.

To allow us to do so, we start with some initial subdivision N_0 (resulting in N_0^4 cubes to examine initially), and let S_0 be the resulting set S of cubes marked as a "yes" or a "maybe". Then, for each four-dimensional cube in S_0 we divide it into N_1^4 cubes, mark each of the smaller cubes as "yes", "maybe", or "no", and let S_1 be the resulting collection of cubes. Note that each cube in S_1 is contained in exactly one cube in S_0. We then repeat the procedure a user-specified number of times. If we choose subdivisions $N_0, N_1, \ldots, N_k$, then the final thickening of X_t is the set S_k. In theory, we should obtain the same thickening of X_t by doing this procedure or by choosing a single subdivision $N_0 N_1 \cdots N_k$.

In practice, however, problems arise if any of the subdivisions $N_0, N_1, \ldots, N_k$ is too coarse. If both the vertex decision algorithm and the Newton's method algorithm mark a face a "no"

that should be a "yes" or "maybe", then not only will that cube not be included in S, but that cube will also not be refined (further subdivided). In other words, we would like to be sure that "no really means no", but we haven't found a way to guarantee this in all cases, although the use of Newton's method certainly helps.

4 Results

We generated six movies (sequences of pictures of a certain projection of X_t for progressively smaller t) that show some interesting behavior. This section analyzes some of that behavior.

For $a = .4$ and $c = .1$, Hubbard expects that J is a solenoid. For $a = .4$ and $c = 1.25$, Hubbard expects that J is a Cantor set. It is not clear what happens to J for $a = .4$ and $c = .5$.

Movie 1 $a = .4$, $c = .1$, t goes from 1.6 to .2 in steps of .1
Projection: xzw, Axes: $[-10, 10] \times [-10, 10] \times [-10, 10]$
Movie 1 demonstrates the behavior of X_t for relatively large t, for these parameter values. Nothing happens except for a general shrinking of X_t.

Movie 2 $a = .4$, $c = .1$, t goes from 1.6 to .2 in steps of .1
Projection: xyw, Axes: $[-10, 10] \times [-10, 10] \times [-10, 10]$
Movie 2 is identical to Movie 1, except that it is a different three-dimensional projection. The coloring suggests that there is no three-dimensional self-intersection (i.e., at the places where it might appear to intersect, the coloring is significantly different, indicating that the values of z are different).

Movie 3 $a = .4$, $c = 1.25$, t goes from 1.9 to .3 in variable steps ($\leq .1$)
Projection: xyz, Axes: $[-8, 8] \times [-8, 8] \times [-8, 8]$
This movie demonstrates the lack of self-intersection (in four dimensions) because of the difference in coloring. Initially, the large outer band "unzips" on each side, but leaving a connected part in the back. Later, this connected part comes apart, only to become reconnected as t gets even smaller, and leaving a "meshing" in the back. Note that X_t is a single connected component throughout the splitting. Then, X_t splits into two connected components, one blue and one black, and each of these components breaks into bands.

Movie 4 $a = .4$, $c = .5$, t goes from .7 to .25 in steps of .05
Projection: xyw, Axes: $[-2, 2] \times [-2, 2] \times [-2, 2]$
Movie 4 demonstrates the problems with using Euler characteristics to analyze X_t. As the surface breaks apart into thin bands, there are always sections of these bands that are very close together (i.e., the size of the gap between them is smaller than the cube size). This could lead to false holes (or falsely filled in holes) in the surface, which could distort the Euler characteristic.

Movie 5 $a = .4$, $c = .5$, t goes from .7 to .25 in steps of .05
Projection: xyz, Axes: $[-2, 2] \times [-2, 2] \times [-2, 2]$

Movie 5 is another projection of Movie 4, illustrating the same band behavior. The false intersection is prominent in the front of the picture.

Movie 6 $a = .4$, $c = 1.25$, t goes from 1.9 to .2 in variable steps ($\leq .1$)
Projection: xyw, Axes: $[-8, 8] \times [-8, 8] \times [-8, 8]$
Movie 6 illustrates the breaking up of X_t into something that will eventually become a Cantor set (because of the parameter values used). As Movie 6 progresses and t gets smaller, X_t breaks up into several connected components. At first, the behavior is similar to that in Movie 3, but then X_t splits in this projection into two separate pieces, each of which then pinch together and split again.

5 Using Cubism

5.1 Running Cubism

Cubism is distributed as a single exectuable file called `cubism`, written in C++ and compiled with g++, which can be run from the command prompt. When `cubism` is run, it reads a file called `args` in the current directory that specifies the parameters for the current run. The format of the `args` file is as follows:

```
a=a_r+a_ii
c=c_r+c_ii
t=t
N=k:N_0,N_1,...,N_{k-1}
x:[x_min,x_max]
y:[y_min,y_max]
z:[z_min,z_max]
w:[w_min,w_max]
UseNewton=u
NewtonIterations=I
NewtonEpsilon=ϵ
```

where

a_r	is the real part of a
a_i	is the imaginary part of a
c_r	is the real part of c
c_i	is the imaginary part of c
t	is the parameter in X_t
k	is the number of subdivision steps
N_i	is the resolution of the ith subdivision.
$x_{\min}$, $x_{\max}$	is the minimum and maximum for the real part of the first complex variable
$y_{\min}$, $y_{\max}$	is the minimum and maximum for the imaginary part of the first complex variable
$z_{\min}$, $z_{\max}$	is the minimum and maximum for the real part of the second complex variable
$w_{\min}$, $w_{\max}$	is the minimum and maximum for the imaginary part of the second complex variable

u is 1 if Newton's Method should be run on the faces (in addition to the vertex decision method), and 0 if only the vertex decision method should be used

I is the number of iterations to be used in Newton's Method.

ϵ is the ϵ used in Newton's Method (a face is counted as a "yes" if after I iterations of Newton's Method, $|G^+ - t| < \epsilon$ and $|G^- - t| < \epsilon$, and the final iteration is within the same face it started in)

For example, if the fourth line of the `args` file read `N=3:30,4,2`, then R would at first be divided up into 30 subdivisions on a side. After that, each cube found in this first level would be divided up into 4 subdivisions on a side. Finally, each cube found in this second level would be divided up into 2 subdivisions on a side. The final cube size would be the same using `N=3:30,4,2` or `N:1:240`.

Note that with the paramters above, Cubism runs with $R = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}] \times [z_{\min}, z_{\max}] \times [w_{\min}, w_{\max}]$.

It is important that the `args` file be specified in exactly the format above, with no missing or extraneous characters (including whitespace). The only exception to this rule is that the last two lines of `args` (i.e., the two lines that give parameters for using Newton's Method) may be omitted if `UseNewton=0`. Note that even if one or both of the parameters a or c are real, it must still be specified in the format above, but with a_i or c_i set to 0. (The same is true if a or c is imaginary.)

On a computer with a Pentium III processor running at about 1 GHz, running Cubism with 3 subdivision steps with $N_0 = 30$, $N_1 = 2$, and $N_2 = 2$ (i.e., with line 4 in the `args` file reading `N=3:30,2,2`) took about 1/2 hour.

5.2 Examining Output

After Cubism is run, it will display a suggested new range (i.e., bounds for R that tightly enclose the approximated surface X_t). If you re-run Cubism with the same parameters, you may want to use the suggested range (assuming the suggested range is smaller than the one that Cubism was run with), because Cubism then will not spend time on cubes that will turn out to be marked as "no"; furthermore, even with the same choices for $N_0, N_1, \ldots, N_{k-1}$, the cubes will be smaller because R is smaller.

Cubism generates several files as output. `cubismresult.m` is MATLAB file that, when run, generates a four-dimensional matrix that contains the approximation of X_t. Each entry of the matrix represents a particular cube, and is 0 if the cube is a "no", 1 if the cube is a "maybe", and 4 if the cube is a "yes". This matrix is not used by any scripts (i.e., it is not needed to plot three-dimensional projections), but it is generated because you may find it useful.

`plotresult.m` is a MATLAB file that, when run, generates eight vectors: `plotx`, `ploty`, `plotz`, `plotw`, `mplotx`, `mploty`, `mplotz`, and `mplotw`. If n denotes the size of `plotx` (the other `plot*` vectors are the same size), then, for each $1 \le i \le n$, (`plotx(`i`)`, `ploty(`i`)`, `plotz(`i`)`, `plotw(`i`)`) is the lower-lower-lower-lower corner of a cube in S that is marked as a "yes". Similarly, if m denotes the size of `mplotx`, then for each $1 \le i \le m$, (`mplotx(`i`)`, `mploty(`i`)`, `mplotz(`i`)`, `mplotw(`i`)`) is the lower-lower-lower-lower corner of a cube in S that is marked

as a "maybe". The companion MATLAB script `cubismplot_single.m` plots a single three-dimensional projection of S, and `cubismplot_multi.m` plots all four three-dimensional projections of S. (The scripts `cubismplot_singlec.m` and `cubismplot_multic.m` create the same plots as `cubismplot_single.m` and `cubismplot_multi.m`, but color the plot using the missing dimension; for example, `cubismplot_singlec(1)` would plot the three-dimensional projection of S into yzw-space, coloring a point (y, z, w) by the missing coordinate x.)

The four files `cubismresult.maple.***` (where `***` is replaced by `xyz`, `xyw`, `xzw`, and `yzw`), are files that are read into Maple (using the companion script `cubismplot.mws`) to plot three-dimensional projections. Sometimes, it is useful to plot projections in Maple, because the projections can be rotated more easily than in Matlab.

5.3 Another Decision Algorithm

One algorithm that we used at one time to try to miss fewer "yes" faces (that were marked "no" by the vertex decision algorithm) is the edge decision algorithm. The idea is to walk along the edges of a face, looking for sign changes in $G^+ - t$ and $G^- - t$. If we find a sign change in both $G^+ - t$ and $G^- - t$, then the face is marked as a "maybe". We later replaced this algorithm with Newton's Method, but the code to do this is still available in the `cubism.cpp` source file. Uncomment the line

```
#define EDGE_DECISIONS
```

at the beginning of `cubism.cpp`. Doing this means that the `args` file must contain the additional line

`SideRes=`s

at the end of the file, where s is the number of steps to make while walking around the edges of a particular face. Turning on edge decisions has no effect on whether or not Newton's Method is used or not.

5.4 The Shell

One debugging tool that we employed was adding a shell of cubes around S_i at each level i to S_i. In other words: after computing S_i using the vertex decision and possibly Newton's Method, we set

$$S_i = S_i \cup \{\text{cubes } c \mid c \notin S_i \text{ but } c \text{ is adjacent to } s \text{ for some } s \in S_i\}. \tag{4}$$

for each i. This means that cubes that we may have missed will be refined anyway. This can be used as a way to prevent us from missing cubes and also as a way to locate the cubes that are being missed. To turn it on, uncomment the line

```
#define USE_SHELL
```

at the beginning of the file `cubism.cpp`.

6 Technical Information

6.1 Narrative

When Cubism is run, the `main` function reads in the parameter values from `args` and calls `do_cubism`. `do_cubism` is responsible for managing the sets of cubes S_i (see section 6.2 for information on how S_i are stored internally) and calling the function `findcubes` to determine which cubes are in S_i, given bounds.

`findcubes` runs through each two-dimensional face within the bounds given and uses the algorithms described in sections 3.1 and 3.2. Each cube is marked with the maximum of the markings of its two-dimensional faces as described in the introduction to section 3.

6.2 Data Structures

Each set of cubes S_i is stored as a hash table with the coordinates of the cubes serving as the keys, and the markings on the cubes ("yes", "maybe") as the values. Cubes with a marking of "no" are not stored. This allows fast access to a cube at a particular coordinate while not storing more in memory than is necessary.

6.3 Source Files

The following source files are needed to compile Cubism. Files with an intial lowercase letter contain global functions and variables that are related; files with an initial uppercase letter are classes.

`cubism.cpp`	The main source file; contains the `main` method.
`henon.cpp`	Contains functions for computing DG^+, DG^-, G^+, G^-, etc.
`output.cpp`	Contains functions for writing Cubism's output files.
`analysis.cpp`	Contains functions for finding connected components and computing Euler characteristics.
`matrix.cpp`	Contains functions for performing matrix operations on two-dimensional arrays.
`utility.cpp`	Contains general utility functions.
`Bounds.cpp`	A class that represents bounds for a particular cube.
`Coordinate.cpp`	A class that represents a cube in $\mathbb{R}^4$.
`Coordinate_d.cpp`	A class that represents a point in $\mathbb{R}^4$.
`CubeList.cpp`	A class that efficiently stores and performs operations on a list of cubes.
`IterableCoordinate_d.cpp`	A subclass of `Coordinate_d` that has special methods built in to work with Newton's method.

6.4 Compiling Cubism

We compiled Cubism using a makefile that is included in the Cubism distribution archive. Cubism relies on the STL as well as the GNU ANSI C++ Library. Each source file is

compiled using `g++` using the `-O3` optimization switch.

A Computing G^+ and G^-

G^+ is computed by attempting to compute the expression inside the limit in the definition of G^+ in (2) for successively larger n until we reach a desired amount of precision (set at 10^{-5}) or until the norm $\left\| H^n\binom{x}{y} \right\|$ gets too large for the computer to handle. G^- is computed by attempting to compute up to $n = 200$, but the computation will terminate early if the norm $\left\| H^{-n}\binom{x}{y} \right\|$ gets too large.

Since any norm can be used to compute G^+ and G^-,

$$\left\| \begin{pmatrix} x \\ y \end{pmatrix} \right\| = N_1 \begin{pmatrix} x \\ y \end{pmatrix} = |x| \tag{5}$$

is used for the computation of G^+ and

$$\left\| \begin{pmatrix} x \\ y \end{pmatrix} \right\| = N_2 \begin{pmatrix} x \\ y \end{pmatrix} = |y/a| \tag{6}$$

is used for the computation of G^-.

B Computing DG^+ and DG^-

To compute DG^+ and DG^-, we consider G^+ and G^- as functions from $\mathbb{R}^4$ to $\mathbb{R}^4$. Let $N_1 : \mathbb{R}^4 \to \mathbb{R}$ be the norm used in the computation of G^+; then,

$$N_1 \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = |x + yi| = \sqrt{x^2 + y^2} \quad \text{and} \quad DN_1 \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{bmatrix} \dfrac{x}{\sqrt{x^2+y^2}} & \dfrac{y}{\sqrt{x^2+y^2}} & 0 & 0 \end{bmatrix}. \tag{7}$$

Let $N_2 : \mathbb{R}^4 \to \mathbb{R}$ be the norm used in the computation of G^-; then,

$$N_2 \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \left| \frac{z + wi}{a} \right| = \frac{\sqrt{z^2 + w^2}}{|a|} \quad \text{and} \quad DN_2 \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{bmatrix} 0 & 0 & \dfrac{x}{|a|\sqrt{z^2+w^2}} & \dfrac{y}{|a|\sqrt{z^2+w^2}} \end{bmatrix}. \tag{8}$$

We compute DG^+ by using the formula

$$DG^+(\vec{v}) = \lim_{n\to\infty} \frac{DN_1(H^n(\vec{v})) \cdot DH(H^{n-1}(\vec{v})) \cdot DH(H^{n-2}(\vec{v})) \cdots DH(H(\vec{v})) \cdot DH(\vec{v})}{2^n \left\| H^n(\vec{v}) \right\|}, \tag{9}$$

where $\vec{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$; similarly, DG^- is computed using the formula

$$DG^-(\vec{v}) = \lim_{n\to\infty} \frac{DN_2(H^{-n}(\vec{v})) \cdot DH^{-1}(H^{-n+1}(\vec{v})) \cdot DH^{-1}(H^{-n+2}(\vec{v})) \cdots DH^{-1}(H^{-1}(\vec{v})) \cdot DH^{-1}(\vec{v})}{2^n \left\| H^{-n}(\vec{v}) \right\|}. \tag{10}$$

To get (9) and (10), we assume that $\log = \log^+$, which is essentially assuming that the iterates of $\vec{v}$ under H and H^{-1} get large. If $\vec{v}$ is in the Julia set, this wouldn't work.

We try to compute the right sides of the above two equations up to a fixed n (we use $n = 10$), or until $x^2 + y^2 + z^2 + w^2 \geq C$, where C is some constant that ensures that we don't encounter an overflow error (we use $C = 10^{50}$).