SOME REMARKS ON LOCAL CONNECTIVITY AT THE FEIGENBAUM POINT

A Dissertation Presented

by

Gregory Benjamin Janks

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Mathematics

Stony Brook University

May 2005

State University of New York

at Stony Brook

The Graduate School

Gregory Benjamin Janks

We, the dissertation committee for the above candidate for the Doctor of Philosophy degree, hereby recommend acceptance of this dissertation.

[illegible]

Scott Sutherland
Professor, Department of Mathematics
Dissertation Director

[illegible]

Mikhail Lyubich
Professor, Department of Mathematics
Chairman of Dissertation

[illegible]

John Milnor
Professor, Department of Mathematics

[illegible]

Saeed Zakeri
Professor, Department of Mathematics, Queens College of CUNY
Outside Member

This dissertation is accepted by the Graduate School.

[illegible]

Dean of the Graduate School

Abstract of the Dissertation

SOME REMARKS ON LOCAL CONNECTIVITY AT THE FEIGENBAUM POINT

by

Gregory Benjamin Janks

Doctor of Philosophy

in

Mathematics

Stony Brook University

2005

We provide a computational approach to proving the Mandelbrot set is locally connected at the Feigenbaum point. By a result of Lyubich, this is equivalent to proving the compactness of an appropriate M-set on the unstable manifold of the fixed point of the renormalization operator. To this end, we establish the necessary rigorous estimates and employ them in conjunction with renormalization methods towards the desired result.

To those who have gone before. And those who have fallen along the way.

Contents

List of Figures

Acknowledgements

First and foremost, I thank Professor Scott Sutherland, not only for guiding me through this project and, in the process, teaching me what computers are all about (it's what the user wants stoopid), but more than that, for demonstrating the effectiveness of a quiet calm during the storm. Scott is all about helping those who can't help themselves, and I was more helpless than most.

My family has provided all manner of support throughout the long winter of this peculiar war. Their love kept me going, and their dollars kept me fed. So, in particular, thanks are due to Daniel and Gerald, and to Mickey and Sadie who saw the project begun, if not completed. My parents, John and Hilary, remain the perfect examples of intellectual curiosity and academic honesty. In more ways than I care to count, this one is for them.

Many people in the Stony Brook Mathematics department, and in its sister Institute, have shown me great kindness. I thank all of them, but particularly, Joann Debis, Donna McWilliams, Lucille Meci, and of course, Gerri Sciulli. I would also like to thank Professor Irwin Kra for his generous attention.

Several mathematicians contributed unselfishly both of their time and wisdom. I would like to thank, in particular, Jack Milnor, Misha Lyubich, Saeed

Zakeri, Oscar Lanford, Rafael de la Llave, Hans Koch, and Doron Lubinsky, all of whom responded personally to my many pleas, and often times provided light in the dark.

Graduate school can be a lonely place. I was most fortunate in my friends: Tom Wehtje, Chris Nagle, Rheo Cushing, the entire racquetball crew, and of course, all the denizens of the House upon the Hill. A particular thanks is due to Mark and Maureen Shaiman who sheltered me quite literally when the writing process spread out over some three hundred miles.

Finally, my wife, Sonia, who can proudly tell you the thesis of this thesis, looked after me, body and spirit, through a process that lasted far longer than anyone foresaw. She remains the essence of me.

Chapter 1

Introduction

Our goal is to prove the Mandelbrot set is locally connected at the Feigenbaum point. To this end, we employ computers to establish many mathematically rigorous quantitative estimates which can be employed in conjunction with renormalization methods towards the desired result. Two small gaps remain in our account, however, which prevent us from claiming complete success. These gaps are completely described, and arguments for their believability are provided.

1.1 Historical Remarks

The Mandelbrot set is truly fascinating. While it is obviously not self-similar, it is densely filled with small copies of itself, and so has something of the flavor of self-similarity. By a result of Lei, the Mandelbrot set is asymptotically self-similar about any Misiurewicz point: zooming in on a Misiurewicz point by a carefully chosen factor acts only as a rotation [Le]. It is also conjectured to be "self-similar" in a slightly different way at the Feigenbaum point [M3].

Considerations of self-similarity in the Mandelbrot set lead inexorably to questions of local connectedness. We quote Douady [D]:

> To what extent are the copies of the Mandelbrot set like the Mandelbrot set? This requires having a description of the Mandelbrot set, so that the copies can be compared to the description. There is a method, Hubbard trees, for describing all the branching information that is known about the Mandelbrot set: what are the braching indexes of the filaments of all the buds on all of the mandelbugs in the Mandelbrot set. Hubbard trees explain all the known images of the Mandelbrot set. It is known that the information given by the Hubbard trees is accurate, but it is not known if it is complete. That is, all the structural information predicted about the Mandelbrot set by the Hubbard trees is correct, but it is not known if Hubbard trees predict all of the structure.
>
> It turns out that the information in the Hubbard trees will be complete if it can be shown that the Mandelbrot set is locally connected at every point.

If the Mandelbrot set is locally connected, we will know a great deal about its combinatorial structure, which will in turn inform us about the bifurcations of quadratic polynomials and many other related maps. "Local connectivity" also implies "Axiom A", the density of hyperbolic dynamics, for quadratic polynomials.

Because of its almost "self-similarity", questions of the Mandelbrot set's local connectivity lend themselves to investigation via renormalization techniques.

These techniques have their origins in the renormalization-group methods of theoretical physics, and were first used in statistical mechanics and quantum field theory. They were adopted by Feigenbaum ([F1], [F2], [F3], [F4]) as he tried to understand how a dynamical system passes from stable to unstable motion, and in particular, to understand the local behavior of a family of dynamical systems in the neighborhood of a parameter value where infinitely many parameter bifurcation values converge.

He continued work begun by Metropolis, M.L. Stein, and P.R. Stein [MSS] who observed that, in the real case, a stable periodic trajectory can become unstable as the parameter value increases, bifurcating into a stable periodic orbit of twice the original period.

Aided by his redoubtable pocket calculator, Feigenbaum noticed the successive parameter values associated with these period doubling bifurcations for the real family $f(x,c) = cx(1-x)$ with $c \in [0,4]$ and $x \in [0,1]$ converged geometrically to $\delta = 4.6692\ldots$. When Feigenbaum found the same constant δ for the family $f(x,c) = c\sin(\pi x)$, he conjectured in 1976 that δ was universal.

Independently, Coullet and Tresser discovered the same "universal scaling law" ([CT1], [CT2]).

To explain this universality, Feigenbaum and Coullet-Tresser conjectured that a specific renormalization operator (which we introduce shortly) has a unique fixed point which is hyperbolic with a one-dimensional unstable manifold.

Although this conjecture was first stated for the period doubling case, it was later extended to include real and complex combinatorics of "bounded type" ([DGP], [GSK]).

This conjecture was first proved by Lanford [L1] with one gap involving a transversality issue later resolved by Eckmann and Wittwer. The key to Lanford's approach was the existence proof of a solution to the *doubling equation* derived jointly by Cvitanović and Feigenbaum ([F1], [Cv]):

$$f(x) = \frac{1}{a} f(f(ax))$$

where $a = f(1)$. This solution can naturally be viewed as the fixed point of the renormalization operator

$$R_{doubl}(f) = \frac{1}{a} \circ f \circ f \circ a$$

where $1/a$ and a denote the obvious multiplication maps. Lanford then proceeded to show that the derivative of this renormalization operator was indeed hyperbolic at the fixed point with a one-dimensional expanding subspace, and with expanding eigenvalue δ. Furthermore, he proved the existence of a parameter value c_{feig} lying between 1.4011550 and 1.4011554 with $\lim_{n\to\infty} R^n(1 - c_{feig}x^2)$ equal to the fixed point of renormalization. We denote this fixed point by *fix*.

The unstable manifold at *fix* was constructed numerically by Vul, Sinai, and Khanin [VSK].

An independent existence proof for *fix* was given by Campanino and Epstein, who later worked with Ruelle ([CE], [CER]). Lanford, himself, provided a second proof based on the Leray-Schauder Fixed Point Theorem [L2], although of these, only Lanford's original proof gave the hyperbolicity of the derivative.

Later, many elements of the Feigenbaum-Coullet-Tresser conjecture were proved without the use of computers. Epstein proved the existence of *fix* ([E1], [E2]). Epstein and Eckmann gave an existence proof of the unstable eigenvalue [EE]. A completely computer-free proof remained elusive, however (the culprits were the transversality and codimension issues). Sullivan provided a program for the construction of *fix* and its stable manifold using Teichmüller theory [S1]. This program was latter executed ([MS], [S2]), while a different approach using geometric limits was used by McMullen [McM2]. The renormalization operator was complexified by Douady and Hubbard [DH].

Finally, Lyubich provided a completely computer-free proof of the Renormalization Conjecture [Ly1] using complex methods. His results are summarised in Section 2.1.

For a good review on Feigenbaum universality, see [VSK]. For an explication of classical renormalization theory, see [CoE], [Cv], and [VSK]. For more recent theory see [MS], [McM1], and [Ly2]. Lyubich also provides an excellent overview running up to his computer-free proof in [Ly1]. For Tresser's historical perspective see [T].

In conclusion, local connectivity of the Mandelbrot set was establised by Yoccoz [Hu] and Lyubich ([Ly3], [Ly4]) for all real values except those which are infinitely renormalizable with bounded type. In 1997, drawing on the work of Yoccoz, Graczyk and Świątek([GS1], [GS2]) and Lyubich [Ly3] independently proved hyperbolic maps are dense in the space of real quadratic maps. Recently, Kozlovski, Shen, and van Strien [KSvS] proved that any real polynomial can be approximated by hyperbolic real polynomials of the same degree.

1.2 Computer assisted proofs

The history of the doubling problem is inherently computational. Since this thesis relies heavily on computational techniques, it seems appropriate to make a few comments.

To start, we quote Lanford [L2] on the matter:

> From a broader point of view, the question of what constitutes a satisfactory proof of an explicit numerical estimate like the one above provides an illuminating caricature of the issues involved in "computer assisted proofs" in general. It hardly seems reasonable to insist that the arithmetic operations be carried out by hand, but relying on results of individual arithmetic operations performed by an electronic calculator does not differ in a fundamental way from relying on results of more complicated sequences of operations performed by a larger computer.

The first hurdle to be cleared in any computational effort is roundoff error. The computer performs computations using floating point arithmetic, and hence has a fixed number of digits of precision. Most arithmetic operations, however, on n-digit operands give answers with more than n-digits. Furthermore, not all rational numbers can be represented by a computer: 0.1, for example, has an infinite binary expansion, and hence no place in the world of bits and bytes.

The solution to this problem is well-known: interval arithmetic. Instead of working with individual numbers, we work with (usually) small intervals containing those numbers, making sure that the endpoints of these intervals are computable. Note that in the complex case, these intervals can be either circular or rectangular. For a more detailed discussion of these issues, see for example [L3].

1.3 Definitions

We denote disks and their boundaries by

$$\begin{aligned} D(z_0, r) &= \{z \in \mathbb{C} \mid |z - z_0| < r\} \\ \overline{D}(z_0, r) &= \{z \in \mathbb{C} \mid |z - z_0| \leq r\} \\ \partial D(z_0, r) &= \{z \in \mathbb{C} \mid |z - z_0| = r\} . \end{aligned}$$

Traditionally, the Mandelbrot set is defined for the family $f_c = z^2 + c$, but following Lanford [L1], we adopt (for the most part) a slightly different parametrization. Let $f_c = 1 - cz^2$, and denote its Julia set $J(f_c)$. Define the *Mandelbrot set* to be the compact set

$$M_0 = \{c \in \mathbb{C} \mid J(f_c) \text{ is connected }\} .$$

In practice, we identify M_0 with its corresponding set of polynomials f_c. Note that $c \in M_0$ if and only if 0 has bounded orbit under f_c.

We write *feig* for the complex function $feig(z) = 1 - c_{feig} z^2 \in M_0$.

Let $\Omega = \{z \in \mathbb{C} \mid |z| < \sqrt{8}\}$. Let $\mathcal{E}_{\mathbb{R}}$ be the Banach space of even functions bounded and analytic on Ω. Let $\mathcal{E}_0$ be the subspace of $\mathcal{E}_{\mathbb{R}}$ consisting of those functions which vanish to second order at 0. Let $\mathcal{E}$ be $\mathcal{E}_0 + 1$. Note that we can write $f \in \mathcal{E}$ as:

$$f(z) = 1 + \sum_{n=1}^{\infty} a_{2n} z^{2n}$$

and that *fix* $\in \mathcal{E}$.

We do not supply a specific norm on this function space. Intuitively, for our purposes, the "norm" needs to be

$$\sup_{z \in U} |f^2(z) - g^2(z)|$$

but it is not not clear what U should be.

1.4 Outline

In Chapter 2, we continue somewhat in survey mode, providing a summary of M. Lyubich's work on Feigenbaum-Coullet-Tresser universality, so we can then provide an outline of our method of proof which depends on the construction of a tube of non-renormalizable functions about *fix*.

In Chapter 3, we detail our method of numerically estimating the unstable manifold of *fix*.

In Chapter 4, we construct analytic criteria which ensure that the functions in the tube about *fix* are not renormalizable.

In Chapter 5, we summarize our computer experiments which verify the established analytic conditions for suitable functions.

In Chapter 6, we detail the two remaining gaps in our proof, and argue for their believability. The first gap involves an analytic estimate of the location of the unstable manifold of fix. The second requires an extension of a renormalization criterion from quadratics to appropriate arbitrary even analytic functions.

Finally, we conclude with two appendices. Appendix A summarizes our computational results, while Appendix B contains the majority of the computer code used in generating them.

Chapter 2

Outline of proof

The method of proof follows a suggestion of M. Lyubich, and depends crucially on several of his results which we summarize here. Full details can be found in [Ly1].

2.1 Results of M. Lyubich (and others!)

Following Douady and Hubbard ([DH]), we define a *quadratic-like map* to be a holomorphic double branched covering (i.e. a proper map of degree 2) between topological disks U and U' such that U is compactly contained in U'. The *filled Julia set* of a quadratic-like map is the set of non-escaping points

$$K(f) = \{z : f^n(z) \in U, n = 0, 1, \ldots\}.$$

Its boundary is called the *Julia* set, and is denoted $J(f)$.

Two analytic maps *represent the same germ at* 0 if they coincide in some neighborhood of 0. By taking all possible analytic continuations, a germ f at the origin can be seen as a full analytic function defined on a Riemann surface covering $\mathbb{C}$. Define $f \sim \tilde{f}$ if they have a common quadratic-like restriction. For the following see [McM1].

Proposition 2.1.1 *Let $f : U \to U'$ and $\tilde{f} : \tilde{U} \to \tilde{U}'$ represent the same germ at 0. Let W be the component of $U \cap \tilde{U}$ containing 0. Then $f \sim \tilde{f}$ iff $0 \in f(W)$. In this case, $g = f|_W$ is a quadratic-like map, and $K(f) = K(g) = K(\tilde{f})$.*

The classes of the equivalence relation $\sim$ are called *quadratic-like germs.* We define $\mathcal{C}$ to be the connectedness locus of the space of quadratic-like germs, i.e. the subset of quadratic-like germs with connected Julia sets.

Now, two quadratic-like maps $f : U \to U'$ and $\tilde{f} : \tilde{U} \to \tilde{U}'$ are *topologically conjugate* if there exists a homeomorphism $h : (U', U) \to (\tilde{U}', \tilde{U})$ such that $h(fz) = \tilde{f}(hz)$ for all $z \in U$. Two germs are *quasi-conformally conjugate* if they admit a quasi-conformal conjugacy. If two maps are qc-conjugate with $\bar{\partial} h = 0$ almost everywhere on the filled Julia set, then they are *hybrid equivalent.*

Take f in the space of quadratic germs. We write $\mathcal{H}(f)$ for the hybrid class of f. The following is due to Douady and Hubbard ([DH]).

Theorem 2.1.2 (Straightening) *If f is a quadratic-like germ with connected Julia set then its hybrid class $\mathcal{H}(f)$ contains a unique quadratic polynomial $P : z \mapsto z^2 + \chi(f)$ where $c = \chi(f)$ is a point of the Mandelbrot set.*

Lyubich showed that the hybrid classes $\mathcal{H}_c, c \in M$ are codimension-1 complex analytic submanifolds of the space of quadratic germs, and that the quadratic family $\{P_c(z) = z^2 + c\}$ is transverse to these submanifolds. Denote by $\mathcal{F}$ the foliation of $\mathcal{C}$ into the hybrid classes. Then:

Theorem 2.1.3 *The foliation $\mathcal{F}$ is transversally quasi-conformal.*

That is to say that the hybrid classes form a codimension-1 quasi-conformal foliation of the space of quadratic germs (see Section 4 of [Ly1]).

Now, Douady and Hubbard also showed ([DH] again) that every hyperbolic component H of the Mandelbrot set M_0 centered at the superattracting parameter value c gives a "*(little) Mandelbrot copy*" M_c which is canonically homeomorphic under some homeomorphism σ to the whole of M_0. If $c \in \mathbb{R}$ then M_c is called *real*, and we note that M_c is symmetric with respect to $\mathbb{R}$, and that if M_c has base period p then its combinatorics are determined by the order of the points 0 and the first p returns of the itinerary of 0 under $z^2 + c$ on the real line. The *root* r_M is the point corresponding to $\frac{1}{4} \in M_0$ under σ. M is termed *primitive* if it is not attached at its root to any other hyperbolic component. Let $\hat{M} = M$ if M is primitive, else $\hat{M} = M \setminus r_M$. We denote by $\mathcal{N}$ the full family of little Mandelbrot copies not including M_0 itself.

Each $M \in \mathcal{N}$ has a locally quasi-conformal *straightening* χ which maps M into M_0. Let $\mathcal{T}_M = \chi^{-1}(M) \subset \mathcal{C}$ be the union of the hybrid classes via M. Similarly, write $\mathcal{T}_{\hat{M}c} = \chi^{-1}(\hat{M}) \subset \mathcal{T}_M$. These $\mathcal{T}_M$ are called *renormalization strips.* If M has period p then the *canonical renormalization operator* R_M on $\mathcal{T}_{\hat{M}}$ is defined as the p-fold iterate of f restricted to an appropriate neighborhood U of the critical point up to rescaling where U is selected so that $f^p|_U$ is a quadratic-like map with connected Julia set, and the "little Julia sets" are pairwise-disjoint except, perhaps, at their β-fixed points. The maps $f \in \mathcal{T}_{\hat{M}}$ are called renormalizable with combinatorics M. Given a family $\mathcal{L} \in \mathcal{N}$ of pairwise disjoint Mandelbrot copies, the operators $R_M, M \in \mathcal{L}$ can be combined in a single operator

$$R_{\mathcal{L}} : \bigcup_{M \in \mathcal{L}} \mathcal{T}_{\hat{M}} \to \mathcal{C}$$

whose restriction to $\mathcal{T}_{\hat{M}}$ is R_M. Given this setup, Lyubich proved:

Theorem 2.1.4 (The QC Theorem) *A primitive copy M of the Mandelbrot set M_0 is quasi-conformally equivalent to the whole of M_0.*

Fix a family $\mathcal{L} \in \mathcal{N}$ of disjoint little Mandelbrot copies and write $R = R_{\mathcal{L}}$. Choose $n \geq 0$. Then a map $f \in \mathcal{C}$ is *n-renormalizable* by R if

$$R^n f \in \bigcup_{\mathcal{M} \in \mathcal{L}} \mathcal{T}_{\hat{M}}, n = 0, 1, \ldots, N-1 \quad R^n(f) \in \mathcal{C}.$$

The itinerary $\tau_N(f)$ of such a map is the sequence $\tau(f) = M_0, M_1, \ldots, M_{n-1}$ of copies $M_j \in \mathcal{L}$ such that $R^j(f) \in \mathcal{T}_{M_j}$.

Let $-\infty \leq l \leq 0 \leq n \leq \infty$. Define an *$(l, n)$-tower* f related to the renormalization operator R as a sequence of quadratic-like germs $f_k : V^k \to U^k$ with connected Julia set $(k = l, \ldots, n)$ such that $f_k = R(f_{k-1})$ on V_k. An infinitely renormalizable germ is said to have *a priori* bounds if the modulus of the annulus $V^n \setminus U^n$ is positive and bounded away from 0 for all n.

Theorem 2.1.5 *Any real infinitely renormalizable quadratic-like germ f with bounded combinatorics has a priori bounds.*

Now let $R = R_M$ denote a renormalization operator with stationary combinatorics $M \in \mathcal{N}$. Write f_* for a *renormalization fixed point* satisfying

$R(f_*) = f_*$. Define the *unstable Mandelbrot set* $\mathcal{M}^u$ as the set of infinitely anti-renormalizable points $f \in \mathcal{C}$ such that $R^{-n}f \to f_*$. Lyubich then proved:

Theorem 2.1.6 (Local unstable manifold) *There exists a complex analytic one-dimensional manifold $\mathcal{W}^u_{loc}(f_*)$ which is transverse to the stable manifold at f_*, is compactly contained in its image under renormalization, and on which the inverse map of renormalization is well-defined.*

This extends as follows:

Theorem 2.1.7 (Global unstable manifold)

1. *A point $f \in \mathcal{C}$ belongs to $\mathcal{M}^u$ iff there exists a one-sided tower $\mathbf{f} = \{f_0, f_{-1}, \ldots\}$ with stationary combinatorics $M, M, \ldots$ and a priori bounds. Moreover, in this case $f_{-n} \in \mathcal{W}^u_{loc}(f_*)$ for all sufficiently large n.*

2. *The straightening $\mathcal{M}^u \to M_0$ is injective.*

3. *For any $\mu > 0$, the set*

$$\mathcal{M}^u_\mu = \{f \in \mathcal{M}^u : \exists \text{ a tower } \mathbf{f} \text{ with } f = f_0 \text{ and } mod(\mathbf{f}) \geq \mu\}$$

   is embedded into a one-dimensional complex analytic manifold $\mathcal{W}^u_\mu(f_*)$ which extends the local manifold $\mathcal{W}^u_{loc}(f_*)$.

4. *The manifold $\mathcal{W}^u_\mu(f_*)$ is transverse to the foliation $\mathcal{F}$.*

5. *The germ of the manifold $\mathcal{W}^u_\mu(f_*)$ near $\mathcal{C}$ is invariant under the renormalization.*

Which gives the following in the case of real combinatorics:

Theorem 2.1.8 *Let $M \in \mathcal{N}$ be a real Mandelbrot set and $R = R_M$ be the corresponding renormalization operator. Then:*

1. *There exists a unique quadratic-like map f_* such that $Rf_* = f_*$, and this map is real.*
2. *The renormalization operator R is hyperbolic at f_*.*
3. *The stable manifold $\mathcal{W}^s(f_*)$ coincides with the hybrid class $\mathcal{H}(f_*)$, and codim $\mathcal{W}^s(f_*) = 1$.*
4. *dim $\mathcal{W}^u_\mu(f_*) = 1$ and the unstable eigenvector λ_* is positive.*
5. *For any $\delta > 0$ there exists a $\mu > 0$ such that the unstable manifold $\mathcal{W}^u_\mu(f_*)$ transversally passes through all real hybrid classes $\mathcal{H}_c$ with $c \in [-2, \frac{1}{4} - \delta]$.*

This finally puts to an end the long journey begun by Feigenbaum, Coulet, and Tresser and continued computationally by Lanford, Eckmann, Wittwer and others, then analytically by Sullivan and McMullen, as Lyubich here provided the first completely non-computational proof which includes the case where c_* is the limit of the period doublings; i.e. $c_* = feig$ and $f_* = fix$.

Our journey is almost at an end. Having done the hard work, Lyubich was able to link the local connectivity of the Mandelbrot set M_0 to the compactness of the M-set in the unstable manifold with the following proposition (proposition 7.5 of [Ly1]).

Proposition 2.1.9 *Let M be a primitive little Mandelbrot set. Let $c_* \in M_0$ be an infinitely renormalizable parameter value of type $M, M, \ldots$ with a priori bounds (for example, a real one). Then the following are equivalent:*

1. *The Mandelbrot set M_0 is locally connected at c_*.*

2. *The unstable Mandelbrot set $\mathcal{M}^u$ of the R_M-fixed point f_* is compact.*

3. *For any $c \in M_0$ there exists a tower $\mathbf{f}_c = \{\ldots \mapsto f_{-1} \mapsto f_0\}$ of type $\{\ldots, M, M\}$ with $\chi(f_0) = c$ and with a priori bounds.*

2.2 Finding a bounded M-set

The crucial import, for our purposes, of Prop. 2.1.9 is that the issue of local connectivity of M_0 at *feig* is translated into a question of compactness of M-sets of *fix*.

A house-keeping issue remains, in that we must specify which little Mandelbrot set M we wish to use. Since our approach is computational, we need to work in a very small neighborhood of *fix* so that our linear approximations (acquired by echoing Lanford) of R_{doubl} obtain; i.e. so that $\mathcal{W}^u_{loc}(\mathit{fix})$ is well represented by the expanding eigenvector of DR_{doubl}.

To this end, we explored neighborhoods of *fix* numerically, in an effort to locate a good candidate M-set. That is to say, an M-set which both appeared bounded, and which was close enough to *fix* so that our numerical estimates applied. This numerical exploration suggested we not work with the standard doubling equation, but instead choose M to be the small copy of base period 4, which intersects the reals at $[-1.4303.., -1.25]$, so that $R_M = R^2_{doubl}$. From this out, as a matter of notation, M and R^2 will refer to these choices. Notice also that, as a matter of definition, $f \in C$ is 4-renormalizable (in the classic sense) iff f is in the domain of R^2.

Notice also that Proposition 2.1.9 is given for primitive components, and we have chosen a satellite component. The corresponding statement applies with the exception that it is not the entire unstable Mandelbrot set which is compact, but only the periodic components.

It remains, then, to prove this compactness. Since $\mathcal{M}^u$ is closed, we need only show that it is bounded. To this end, we construct a Jordan curve in $\mathcal{W}^u_{loc}(\mathit{fix})$ which compactly contains $\mathcal{M}^u$ in its interior. Since M is a real Mandelbrot set, $\mathcal{M}^u$ is symmetric in the real axis, and so we need only work with functions whose coeffecents have (say) positive imaginary values. Now, the ray with external angle $2/9$ lands at the root point of $\mathcal{M}^u$, and the functions on this ray cannot be 4-renormalizable. So the problem reduces to building a Jordan curve in $\mathcal{W}^u_{loc}(\mathit{fix})$ connecting this ray to the real axis while avoiding $\mathcal{M}^u$. This we do, essentially by computation. Following Lanford, we can approximate fix, at least in some sense, to arbitrary precision, and so can approximate $\mathcal{W}^u_{loc}(\mathit{fix})$ by computing the expanding eigenvalue of DR and then applying R to the corresponding eigenvector. Since we are close to fix, this approximation must be good. We can then compute a large (but obviously finite) number of honest polynomials approximating the required arc in $W^u_{loc}(\mathit{fix})$, and with these polynomials in hand, we can check by direct computation to see if they are 4-renormalizable.

Of course, we have to account for errors in this approximation. To this end, we deliver strict analytic requirements, and perform all computations using interval arithmetic so that there is no possibility of round-off error. Finally, by using interval arithmetic, we are able to "fatten" our choice of polynomials into a "tube" based on our approximation of $\mathcal{W}^u_{loc}(fix)$, the idea being that this tube intersects the actual $\mathcal{W}^u_{loc}(fix)$, thus establishing the result.

Chapter 3

The Unstable Manifold

We use computational estimates rather than analytic arguments to estimate a parametrization of the unstable manifold $\mathcal{W}^u_{loc}(\mathit{fix})$. This is the first gap mentioned in Chapter 1. Arguments for the believability of our methods can be found in Chapter 6.

Since arbitrary elements of $\mathcal{W}^u_{loc}(\mathit{fix})$ are analytic, they are well represented by Chebyshev approximations. That is to say that we can represent the action of an arbitrary element in $\mathcal{W}^u_{loc}(\mathit{fix})$ on the z-plane by

$$\sum_j \alpha_j T_{2j}(z)$$

where the $T_{2j}(z)$ denote standard Chebyshev polynomials. Chevyshev polynomials are used for two reasons: one, they have nice truncation properties, and two, they are much easier to curve fit than standard polynomials.

The resulting α_j can in turn be approximated by Chebyshev polynomials of their own, to provide a parametrization of $\mathcal{W}^u_{loc}(\mathit{fix})$ itself using a one-complex dimensional parameter variable t.

This gives our parametrization of $\mathcal{W}^u_{loc}(\mathit{fix})$ as a double power series $W^i(t,z)$ with

$$W^i(t,z) = \sum_{j,k} b^i_{k,j} T_k(t) T_{2j}(z).$$

Note that we always set $W(0,z) = \mathit{fix}$, and choose $W(1,z)$ so that M will be compactly contained in the region of the unstable manifold parametrized by $W(t,z)$ with $|t| < 1$, but not so large we lose numerical control.

Within this framework, we develop an iterative procedure which produces successively better approximations W^i of $\mathcal{W}^u_{loc}(\mathit{fix})$. However, since the reals are preserved and our analytic functions extend in a unique way, and since Chebyshev curve-fitting techniques only require computations on [0,1], we can restrict our attention to this real interval.

Obviously, the unstable eigenvector v provides a good initial approximation, so we set

$$W^0(t,z) = [\mathit{fix} + t * \sigma * v](z)$$

where σ is an appropriate scaling factor (we use $\sigma = 0.001$) ensuring we parameterize only the region of interest and retain numerical control.

Now, given an approximation W^i we compute W^{i+1} as follows.

Intuitively, if λ denotes the expanding eigenvalue, then

$$\mathcal{W}^u_{loc}(\mathit{fix}) \approx \lim_{n\to\infty} \frac{R^n_{doubl}(W^0)}{\lambda^n}$$

with this equality holding over our region of interest. This suggests setting

$$W^{i+1} \approx \frac{R^n_{doubl}(W^i)}{\lambda^n}$$

for some reasonably large n (in this computational context $n = 2$ is "reasonably large"). It remains only to define this "division" operator.

The purpose of the division operation is to prevent our losing numerical control. That is to say we want the region of the unstable manifold which we are parameterizing to remain relatively constant across iterations.

But the scale of this region was determined by our choice of W^0. So, given W^i, define $t_{i+1} \in [0, 1]$ to be $\lambda^{-n} \cdot \pi(W^i(1))$ where π denotes projection onto v. Now define our iterative procedure (for a suitable n) by:

$$W^{i+1}(t, z) = R^n_{doubl}(W^i(t \cdot t_{i+1}, z)).$$

Since R^n_{doubl} acts as expansion by λ^n in the v direction, we can be sure that computationally

$$\lambda^n \cdot \pi(W^{i+1}(t_{i+1})) \approx \pi(W^i(1))$$

and that our region is indeed comparable across iterations (if there were no computational or approximation issues the equality would be exact).

Chapter 4

The Estimating Lemma

4.1 Notation

For $f \in \mathcal{E}$, we write:

$$\begin{aligned} a_f &= f^2(0) \\ b_f &= f^3(0) \\ c_f &= f^4(0) \end{aligned}$$

and $per2_f$ for the period 2 point of f with smallest modulus.

4.2 Renormalization Criterion

The first step is to obtain a criterion for renormalizability by R^2. Notice, that a function $f \in \mathcal{E}$ is 4-renormalizable if there exists a closed topological disk Δ in the z-plane such that:

- $f^4(\Delta) \supset \Delta$ with $f : \Delta \to f^4(\Delta)$ a degree 2 proper map

- $f^n(0) \in \Delta$ if $n \bmod 4 = 0$
- $f^n(0) \notin \Delta$ if $n \bmod 4 \neq 0$.

We posit the following hypothesis, but only supply an outline of the proof. This is the second gap mentioned in Chapter 1.

The conjecture was formulated after numerically mapping honest circles of varying radii away from and back over zero in the appropriate way. These experiments focussed attention on the return time of the period 2 point with smallest modulus. Since the functions involved are somewhat constrained – they all lie in the period 2 wake for example – examining their individual Julia sets revealed that those which met the following criterion all had the property of being "wider than they were tall". It is this key fact which motivates the conjecture.

Conjecture 4.2.1 *Let $f \in \mathcal{E}$ be near to fix (coefficent-wise). If $|c_f| > |per2_f|$ then f is not R^2-renormalizable.*

4.2.1 Outline of conjecture's proof

Instead of working with R^2 and $|per2_f|$, we may, without loss of generality, reduce the problem as follows. Let $f \in \mathcal{E}$ be near to *fix* (coefficent-wise). Let $\alpha(f)$ denote the α-fixed point of f. If $|a_f| > |\alpha(f)|$ then f is not R_{doubl}-renormalizable.

Our general approach is to prove the lemma for quadratics near *fix* (which is to say quadratics near *feig*), then to extend this for higher order terms, all but finitely many of which can be estimated analytically.

Lemma 4.2.2 *Let $f(z) = z^2 + a$ with $|a - c_{feig}| < \frac{3}{4}$. Then $|a_f| > |\alpha(f)|$ implies f is not R_{doubl}-renormalizable.*

Proof:

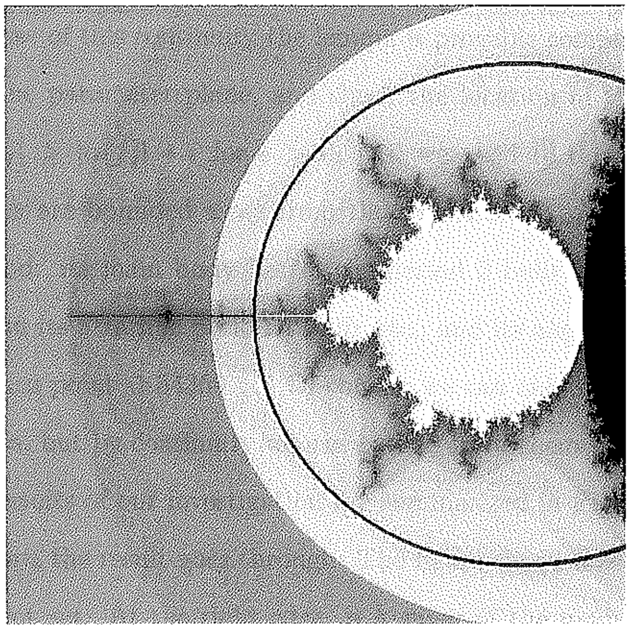

Figure 4.2.1: Quadratics with $|a_f| > |\alpha(f)|$ are bounded away from the M-set of R_{doubl}.

As will become our modus operandi, our method of proof is brute force computation. For an arbitrary quadratic f near $feig$, we need to bound the amount the "little Julia set" of f can extend past $\alpha(f)$. If f is R_{doubl}-renormalizable, there exists a topological disk containing 0 which is mapped over itself by $R_{doubl}f$, and this disk must contain the "little Julia set", and hence a_f. So if $|a_f|$ is indeed bounded away from $|\alpha(f)|$, then either f is in fact not R_{doubl}-renormalizable, or the "little Julia set" is a Cantor set, and thus disconnected, which means f cannot be in the M-set of R_{doubl}. This is exactly because of their being "wider than they are tall".

Now, for a given f we can establish a set which covers its Julia set using standard techniques (checking escape times from a disk of radius 2). Moreover, this covering set varies continuously in the Hausdorff metric with respect to the parametrization of f.

Because of this continuity the lemma is provable computationally. We can inspect the parameter plane, and check the location of those quadratics for which $|a_f| > |\alpha(f)| + \epsilon$ for $\epsilon > 0$. All we need to do is verify that these quadratics are bounded away from the M-set of R_{doubl}. This M-set is well understood: its main body lies in the period 2 bulb of the Mandelbrot set, and within its attached period 4 bulb, etc., and its antenna terminates at the Misiurewicz point -1.54369.... Of course, we can only examine finitely many quadratics, but by continuity we can extend from them to the desired result.

The results of our investigation are summarized in Figure 4.2.1.

In short, the result goes through. The figure shows a zoomed image of the Mandelbrot set. At the right edge, is a sliver of the main cardioid. Next to this is the period 2 bulb. The M-set of R_{doubl} is highlighted, and surrounded by three rings which contain all quadratics for which $|a_f| > |\alpha(f)| + \epsilon$ with $\epsilon \geq 0.2$ in the outer band, $0.01 \leq \epsilon < 0.2$ in the middle one, and $0 \leq \epsilon < 0.01$ in the inner. The leading edge of the inner ring has $\epsilon = 0$, and intersects the x-axis at the Misiurewicz point -1.54369.... The width of the inner band is 0.006, and the width of the middle band is 0.102, with both measurements taken along the x-axis. $\square$

Given this, the only missing element required to prove Conjecture 4.2.1 is an extension to arbitrary functions in $\mathcal{E}$ close to fix using standard Taylor type arguments. The rate of decay of *fix*'s higher order terms is well known, and quite rapid, so while this extension may require some technical prestidigitation, its veracity is practically certain.

4.3 The Estimating Lemma

Continuing, our fundamental approximating tool is the following invariant form of the *Koebe* $1/4$ *theorem*.

Theorem 4.3.1 (Koebe) *Let ψ be a conformal mapping from a simply connected domain Ω_1 onto a simply connected domain Ω_2, and let $\psi(z) = w$. Then*

$$\frac{|\psi'(z)|}{4} \leq \frac{\operatorname{dist}(w, \partial\Omega_2)}{\operatorname{dist}(z, \partial\Omega_1)} \leq 4|\psi'(z)|.$$

And, in particular, if $f : D(z,r) \to \Omega_2$ is conformal with Ω_2 simply connected, then

$$\frac{r|f'(z)|}{4} \leq \operatorname{dist}(w, \partial\Omega_2) \leq 4r|f'(z)|.$$

Suppose we have an explicit polynomial $f \in \mathcal{E}$. By direct computation, we find a good approximation of f's appropriate period two point, and call this approximation $per2^*_f$, noting that $|f^2(per2^*_f) - per2^*_f|$ is small. We can then compute $|c_f| - |per2^*_f|$ exactly. To deal with an arbitrary g "close" to f (i.e. to compare $|c_g|$ with $|per2_g|$), we divide the problem as follows.

Claim 4.3.2 (Organizing) *If*

$$|per2^*_f - per2_f| \quad < \quad A$$

$$\begin{aligned} |c_f - c_g| &\leq B \\ |per2_f - per2_g| &< C \\ |c_f| &= |per2^*_f| + E \end{aligned}$$

then

$$|c_g| > |per2_g| + E - (A + B + C).$$

Proof: Immediate: $|c_g| - |per2_g| > |c_f| - B - (C + |per2_f|) > |per2^*_f| + E - B - C - (A + |per2^*_f|)$. □

This Organizing Claim provides a clear line of attack. For a given polynomial f:

- Compute a good approximate period 2 point.
- Compare its modulus with that of c_f. This gives us a "surplus" for errors in our approximations.
- Compute a bound on the difference between the approximate period 2 point and the actual period 2 point.
- Compute a bound on the difference between $per2_f$ and $per2_g$.
- Compute a bound on the difference between c_f and c_g.
- Subtract these three bounds from the surplus and check that the result is positive.

The following standard result helps resolve the approximate period 2 point (see, for example, [H]).

Theorem 4.3.3 *Let p be a complex polynomial of degree d and let z_0 be a point s.t. $p'(z_0) \neq 0$. Let h_0 denote the Newton correction of p at z_0. Then any circular region with the point z_0 and $z_0 + dh_0$ on its boundary contains at least one zero of p.*

Corollary 4.3.4 *Let $f \in \mathcal{E}$ be a polynomial of degree n. Suppose there exist $\alpha, \delta > 0$ and $per2^*_f \in \mathbb{C}$ such that*

$$\begin{aligned} |f^2(per2^*_f) - per2^*_f| &< \delta \\ |(f^2)'(per2^*_f) - 1| &= \alpha. \end{aligned}$$

Then writing

$$A = \frac{\delta n^2}{\alpha}.$$

gives $per2_f \in D(per2^*_f, A)$. That is

$$|per2^*_f - per2_f| < A.$$

Proof: Consider the complex polynomial

$$p(z) = f^2(z) - z$$

of degree n^2 in Theorem 4.3.3 with $z_0 = per2^*_f$. Certainly,

$$|p'(per2^*_f)| = \alpha > 0,$$

and the Newton correction h_0 of p at z_0 satisfics

$$|h_0| = \left|\frac{p(per2^*_f)}{p'(per2^*_f)}\right| = \left|\frac{f^2(per2^*_f) - per2^*_f}{(f^2)'(per2^*_f) - 1}\right| < \frac{\delta}{\alpha}.$$

The disk $D(per2^*_f, A)$ certainly contains the the circle whose center lies on the midpoint between $per2^*_f$ and $per2^*_f + n^2 h_0$, and whose boundary therefore contains both points. Applying the theorem gives the result. □

Proposition 4.3.5 *Let $f, g \in \mathcal{E}$. Choose $M_1, M_2 > 0$ such that*

$$\begin{aligned} |a_f - a_g| &< M_1 \\ |(f^2 - g^2)(a_g)| &\leq M_2. \end{aligned}$$

Suppose also that neither 0 nor $f^{-1}(0)$ are in $D(a_f, M_1)$, and write

$$B = 4M_1|(f^2)'(a_f)| + M2$$

Then

$$|c_f - c_g| \leq B.$$

Proof: Write $F = f^2$ and $G = g^2$, so that F is conformal on $D(a_f, M_1)$. The proposition then becomes a simple application to the Koebe 1/4-theorem to F on this disk. Since

$$G(0) = g^2(0) = a_g \in D(a_f, M_1),$$

Koebe says

$$|F(F(0)) - F(G(0))| \leq 4M_1|F'(F(0))|.$$

But then

$$|F(F(0)) - G(G(0))| \leq |F(F(0)) - F(G(0))| + |F(G(0)) - G(G(0))|$$

i.e.:

$$\begin{aligned} |c_f - c_g| &= |f^4(0) - g^4(0)| \\ &= |F(F(0)) - G(G(0))| \\ &\leq 4M_1|F'(F(0))| + |F(G(0)) - G(G(0))| \\ &\leq 4M_1|(f^2)'(a_f)| + M_2. \end{aligned}$$

□

Finally, we turn our attention to $|per2_f - per2_g|$.

Proposition 4.3.6 *Let F and G be complex analytic functions. Suppose there exists $r, \epsilon, M > 0$ and $z_F \in \mathbb{C}$ such that:*

$$\begin{aligned} F(z_F) &= 0 \\ r &> 8M/\epsilon \\ |F'(z)| &> \epsilon \quad \forall\, z \in D(z_F, r) \\ |F(z) - G(z)| &< M \ \ \forall\, z \in \overline{D}(z_F, r). \end{aligned}$$

Then, there exists $z_G \in D(z_F, r)$ with $G(z_G) = 0$.

Proof: Since F is conformal on $D(z_F, r)$, we may apply Koebe to prove

$$F(D(z_F, r)) \supset D(0, \frac{r|F'(z_F)|}{4}) \supset D(0, 2M)$$

i.e.

$$\mathrm{dist(F(\partial D(z_F, r)), 0) > 2M.}$$

But

$$|F(z) - G(z)| < M \text{ on } \overline{D}(z_F, r),$$

hence

$$\mathrm{dist(G(\partial D(z_F, r)), 0) > M.}$$

Since G is analytic, $G(D(z_F, r))$ is either wholly "inside" $G(\partial D(z_F, r))$ or wholly "outside". But

$$|G(z_F)| = |F(z_F) - G(z_F)| < M.$$

Therefore $0 \in G(D(z_F, r))$; i.e. there exists $z_G \in D(z_F, r)$ with $G(z_G) = 0$. $\square$

Now, we apply this to the periodic case. For the purposes of calculation, we want the result on a disk centered at $per2^*_f$.

Corollary 4.3.7 *Let $f, g \in \mathcal{E}$ with f a polynomial of degree n. Suppose there exists $\epsilon, \delta, A, M_3, C > 0$ and $per2^*_f \in \mathbb{C}$ such that*

$$\begin{aligned} A &= \frac{\delta n^2}{\epsilon} \\ |f^2(per2^*_f) - per2^*_f| &< \delta \end{aligned}$$

$$
\begin{aligned}
C &> 8M_3/\epsilon \\
|f^2(z) - g^2(z)| &< M_3 \quad \forall\, z \in \overline{D}(per2^*_f, A + C) \\
|(f^2)'(z)| &> 1 + \epsilon \ \ \forall\, z \in D(per2^*_f, A + C)
\end{aligned}
$$

Then, $|per2_f - per2_g| < C$.

Proof: By corollary 4.3.4 we have $per2_f \in D(per2^*_f, A)$, and hence $D(per2_f, C) \subset D(per2^*_f, A + C)$, so that the inequalities on the larger disk also hold on the smaller one. Write

$$
\begin{aligned}
F(z) &= f^2(z) - z \\
G(z) &= g^2(z) - z \\
z_F &= per2_f \\
M &= M_3 \\
r &= C
\end{aligned}
$$

in proposition 4.3.6 to prove the existence of $per2_g \in D(per2_f, C)$ such that $G(per2_g) = 0$; i.e. $g^2(per2_g) = per2_g$, and $|per2_f - per2_g| < C$. □

Combining these results proves the *Estimating Lemma*.

Lemma 4.3.8 (Estimating) *Let $f, g \in \mathcal{E}$ with f a polynomial of degree n. Suppose there exists $\epsilon, \delta, A, B, C, E, M_1, M_2, M_3 > 0$ and $per2^*_f \in \mathbb{C}$ such that*

$$\begin{aligned}
A &= \frac{\delta n^2}{|(f^2)'(per2^*_f) - 1|}\\
B &= 4M_1|(f^2)'(a_f)| + M_2\\
C &> 8M_3/\epsilon\\
E &= |c_f| - |per2^*_f|\\
|f^2(per2^*_f) - per2^*_f| &< \delta\\
|a_f - a_g| &< M_1\\
|f^2(a_g) - g^2(a_g)| &\leq M_2\\
|f^2(z) - g^2(z)| &< M_3 \quad \forall\, z \in \overline{D}(per2^*_f, A+C)\\
|(f^2)'(z)| &> 1+\epsilon \ \ \forall\, z \in D(per2^*_f, A+C)
\end{aligned}$$

and neither 0 nor $f^{-1}(0)$ are in $D(a_f, M_1)$, then

$$|c_g| > |per2_g| + E - (A+B+C).$$

and hence g is not 4-renormalizable if $E > A+B+C$.

The inequalities involving M_1, M_2, and M_3 reveal where the "norm" $\|f-g\|$ needs to be evaluated: precisely at 0, a_g, and, most problematically (at least for the notion of a norm), on a disk about $per2^*_f$.

Chapter 5

The Computation

5.1 Overview

The heart of our proof is a massive computational effort to verify the Estimating Lemma on an arc in $\mathcal{W}^u(\mathit{fix})$. All of the calculations are performed in C++, using both an arbitrary precision package called CLN, and a complex interval arithmetic package called CXSC.

The method is as follows. We parameterize an arc of 1,000 points on our approximation to $\mathcal{W}^u(\mathit{fix})$. These functions are large degree polynomials (we use degree 50), represented by their coefficents. Given two such functions, we can therefore generate the complex interval convex hull containing them by performing the hulling operation coefficent-wise.

We pick two successive functions on the arc, and generate the convex hull g containing them. We set f in the Estimating Lemma to be the center of this convex hull, and establish that the lemma holds for f and g computationally (and thus that it holds for all functions in the hull).

The first order of business is to find a suitable $per2^*_f$. To this end, we use

CLN and an adapted "False Position" method to find $per2^*_f$ satisfying

$$|f^2(per2^*_f - per2^*_f| < 10^{-50}$$

and then, once we have the candidate, we use interval arithmetic to verify its suitability.

We can then compute M_1, M_2, δ, A, B, C, and E all directly from the formulae in the Estimating Lemma. M_3 and ϵ, however, require some work. Once we have them, though, it is a simple matter of interval computation to verify the positivity of the appropriate quantities and to check the location of $f(0)$ and $f^{-1}(0)$.

5.2 Choosing ϵ

Following a suggestion of D. Lubinsky, we use an inequality of Duffin-Shaeffer type due to Frappier, Rahman, and Ruscheweyh [MMR]:

Theorem 5.2.1 *Let P be a polynomial of degree n. Then the maximum modulus of P' over the unit circle is bounded above by n times the maximum of P at the $2n$-th roots of unity.*

Corollary 5.2.2 *Let P be a polynomial of degree n. Let L be the maximum modulus of P at the $2n$-th roots of unity. Let M be the minimum modulus of P at the $2n$-th roots of unity. Let m be larger than $2n$. Then, by estimating the difference between P at the roots of unity and at any point on the unit circle, we have*

$$min\{|p(z)| : z \text{ on unit circle } \} = M - \pi L\frac{n}{m}$$

In practice, we pick some small radius r (remembering later to verify $A + C < r$; for completeness we use r=0.014), and set

$$\begin{aligned} \phi(z) &= rz + per2^*_f \\ P(z) &= (f^2)'(\phi(z)) \end{aligned}$$

so that ϕ maps the unit circle onto $D(per2^*_f, r)$. We now need to bound $|P|$ from below on the unit circle, and the corollary applies, so we evaluate P at the $2n$-th roots of unity and compute M. Then we choose an integer m much larger than n, and calculate L.

5.3 Choosing M_3

Recall that we are trying to bound

$$|f^2(z) - g^2(z)|$$

from above. We use an application of Stečkin's Lemma due to Green [G].

Lemma 5.3.1 (Stečkin) *Suppose that the polynomial*

$$f(t) = \sum_{|n| \leq N} \alpha_n e^{int} \text{ with } \alpha_n \in \mathbb{C}$$

is real-valued and that $t_0 \in [0, 2\pi]$ satisfies $f(t_0) = \|f\|_\infty$ where

$$\|f\|_\infty := \sup\{|f(z)| : z \in \mathbb{C}, |z| \leq 1\}.$$

Then

$$f(t_0 + s) \geq \|f\|_\infty \cos Ns \quad \textit{for } |s| \leq \frac{\pi}{N}.$$

Again set

$$\phi(z) = rz + per2^*_f$$

mapping the unit circle onto $D(per2^*_f, r)$ with r suitably chosen as above and then use

$$\begin{aligned} P(z) &= f^2(\phi(z)) - g^2(\phi(z)) \\ Q(t) &= |P(e^{it})|^2 \end{aligned}$$

so that the problem is translated into real terms. Then, following Green, we write $N = \deg(Q) = \deg(P) = \deg(f^2) = \deg(g^2)$, and divide $[0, 2\pi]$ into n equal intervals $I_1, ..., I_n$, choosing $n > 2N$, each with midpoint t_k and width $2h$, so that $h < \frac{\pi}{N}$. If a global maximizer t_0 is in $I_k = [t_k - h, t_k + h]$ then by Stečkin

$$Q(t_k) \geq \|Q\|_\infty cosN(t_k - t_0) \geq \|Q\|_\infty \cos Nh.$$

Thus

$$\|Q\|_\infty \leq Q(t_k) \sec Nh$$

and, in particular, if $\tilde{Q} := \max_{i=1,...,n}\{Q(t_i)\}$ then

$$\tilde{Q} \leq \|Q\|_\infty \leq \tilde{Q} \sec Nh$$

but each of the $Q(t_i)$ (and hence $\tilde{Q}$), N, and h are all computable.

5.4 Results

Chebyshev polynomials were used to record the coefficents of the fs in case of possible truncation issues. This entailed a reproduction of Lanford's efforts to find *fix* in our own coordinate system. CLN was again used with 50 digits of precision in the mantissa. With *fix* in hand, we approximated DR, the derivative of R_{doubl}-renormalization, at *fix*, and used Maple to derive an estimate for R's expanding eigenvector, again in Chebyshev coordinates.

These approximations are summarized by Tables A.1 (showing *fix*) and A.2 (showing the eigenvector) in Appendix A.

Finally, Table A.3 provides a summary of the Estimating Lemma values for M_1, M_2, M_3, δ, ϵ, A, B, C, E, and $E-(A+B+C)$ for representative fs on the arc. In short, the experiment is succesful, and the Estimating Lemma goes through.

Figure 5.4.1 provides some insight into the behavior of $|per2^*_f|$ compared to that of $|c_f|$. As the picture moves from left to right, the angle moves from $\frac{\pi}{2}$ to 0 where f is purely real.

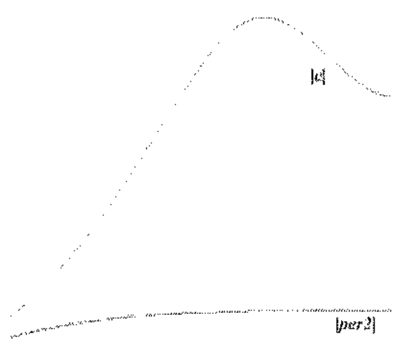

Figure 5.4.1: The graph of $|c_f|$ is clearly above that of $|per2^*_f|$.

Figure 5.4.2 is a computer approximation of the M-set of *fix* on $\mathcal{W}^u(\mathit{fix})$. It is based at the computer's approximation of *fix*. Other "points" represent functions parametrized by our approximation of $\mathcal{W}^u(\mathit{fix})$. Each point is colored based on its (approximate) polynomial-like behavior: depending on whether or not its critical point returns to the disk of radius $|per2^*|$ for return times which are multiples of 4, and for no other. It is indeed a somewhat familiar shape.

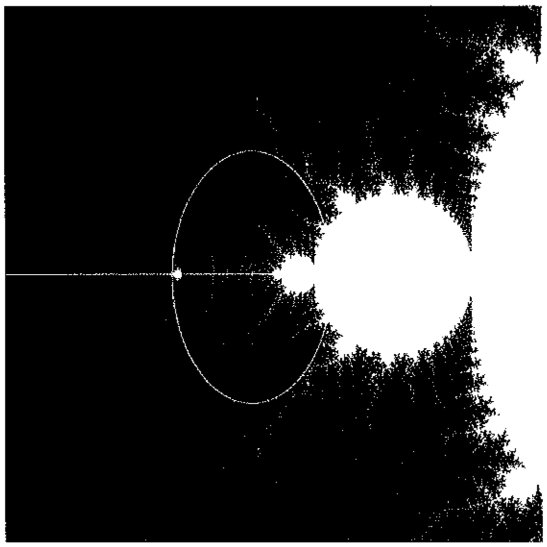

Figure 5.4.2: A computer approximation of the M-set of *fix* on $\mathcal{W}^u(\mathit{fix})$. The restricted M-set used in the Estimating Lemma is contained in the ellipse which is included for illustrative purposes only.

Chapter 6

Conclusion

Throughout, our goal has been to prove the Mandelbrot set is locally connected at the Feigenbaum point. The work of M. Lyubich reduces this problem to constructing a Jordan curve in $\mathcal{W}^u_{loc}(\mathit{fix})$ which compactly contains $\mathcal{M}^u$ in its interior, and this we have done, using computational methods.

Two important gaps, however, remain.

The first is an analytic proof that the computational process detailed in Chapter 3 converges quickly enough to the unstable manifold $\mathcal{W}^u_{loc}(\mathit{fix})$. That such a proof exists is not unreasonable. Certainly, the process converges to the unstable manifold by nature of its construction: we start with an excellent approximation v of the unstable eigenvector based at *fix* and given in Table A.2. We renormalize several times, then divide appropriately by the unstable eigenvalue.

Computational exploration of this procedure suggests that in our particular small neighborhood of *fix*

$$|\mathcal{W}^u - v| < 2.6e^{-7}$$

where the norm is taken coefficent-wise. This estimate is, of course, limited by our computational assumptions; we are after all "only" using fifty coefficents to fifty places so, at the end of the day, estimating the norms of our two-dimensional approximations of $\mathcal{W}^u$ are probably not much good past roughly ten decimal places. Since our approximations are (probably!) very accurate this practical limit is very quickly reached. Of course, this is all somewhat arbitrary. There is certainly no theoretical limit preventing us from sharpening the estimate to essentially arbitrary precision. For us, the problem was somewhat exacerbated, because the available implementation of CXSC did not allow for arbitrary precision in interval computations, but this again is a limitation of circumstance, and not of approach. Finally, we note that the size of the convex hulls used in Section 5 are roughly double that of the distance of $2.6e^{-7}$ suggested by numerical exploration, so while this doesn't prove anything, it suggests we have either captured the unstable manifold, or we are extremely close to it. Once the precise location of $\mathcal{W}^u$ is ascertained, another run of the Estimating Lemma computation may be required, but this is highly unlikely to effect the outcome, as the computational process has proved itself incredibly robust in a large (relative to the scale) neighborhood of the data provided here.

In going forward, there are of course, valid theoretical approaches available to verify the location of the unstable manifold.

The contracting eigenvalues of renormalizaton can almost certainly be rigorously bounded over a ball about fix, and this, in conjunction with a suitable cone field argument, should provide the necessary estimate. Lanford, in a private communication, writes of these contracting eigenvalues:

I've never tried to prove any. I don't imagine there would be any particularly difficulty in doing this, but it would almost certainly require attention to a great many uninteresting details, and I have never felt a great enough need for such bounds to justify the effort it would cost to obtain them. If you mean accurate numerical values: Some 15 years ago, Henri Epstein and I made a kind of a sport of high-accuracy computation of things related to the Feigenbaum fixed point - and in particular of the eigenvalues ... their accuracy has been tested empirically in a variety of ways. At least for the first half dozen of them, nearly all digits shown are almost certainly right, and all of them are almost certainly accurate to 20 decimal places or more.

Lanford's estimates are provided in order of decreasing magnitude in Table A.4.

An alternative approach would be to directly generate a bound on the second derivative of renormalization at *fix*, possibly in a program similar to Lanford's original one.

The second gap in our proof is the missing element in our proof of Conjecture 4.2.1, our criterion of renormalizability: the extension of the result to arbitrary functions in $\mathcal{E}$ close to *fix*. For the reasonableness and viability of this see (the end of) Section 4.2.

So, while we have not quite achieved our goal, we are certainly very close!

Appendix A

Computational Results

A.1 Approximating *fix*

Coefficent	*Value*
T_0	0.2828954316362471577526617239876760110621069047279353041729
T_2	-0.7003915739737137871459804403324712617270497768778257557539
T_4	0.01736218672224418420927870994000841793524031574731849968944
T_6	6.236559136940908585298153810290290353837554147664859947748e-4
T_8	-2.5266414376208310293595863274053777214179087320600289998064e-5
T_{10}	2.781260604292479399098215157081793498586841984830573789424e-7
T_{12}	7.7936819963404276117862621362957196474078095719352923752e-9
T_{14}	-3.2778572258673012387370875937161509886780590318970900485026e-10
T_{16}	6.384210078705431885183925426750487068395886514077068542389e-13
T_{18}	1.7781071145945873454983293188050153549710969654409014365472e-13
T_{20}	-2.7990431550152434050830449792221265830047852261480888082276e-15
T_{22}	-6.598776570642374208242061975960948460560392623237023675573e-17
T_{24}	3.1157462047608457677156074800260627137687788878386060507025e-18
T_{26}	-2.6288817352565470028780281587289265745294024346924213473363e-20
T_{28}	-1.3261557529235466453727845429887678725640048406244467016826e-21
T_{30}	4.568921328589640700367237904701758560955651483669949156935e-23
T_{32}	-2.616691438427295430525032619729071167561350995479885278421e-25
T_{34}	-1.9483941714088536304968295816829343632476027547185760338803e-26
T_{36}	5.488618268000511087819691527845096363647944167413544759342e-28
T_{38}	-8.036151191177115413983222267610295111168691526427775512693e-31
T_{40}	-2.7292878323462550963098446174037920692352661046818821171747e-31
T_{42}	5.580280767578030698966965128999181747442177896966392624412e-33
T_{44}	4.285793688386266970351157117604517047558091566901953814854e-35
T_{46}	-3.805534639185068232693473343394153263837292021698723365881e-36

Table A.1, Approximation of fix

Coefficent	*Value*
T_{48}	4.547280331539462192806400845558070858956256439228764035598e-38
T_{50}	1.3345353991736648682921149474045776882693633744857054591495e-39

Table A.1: Approximation of *fix*

A.2 Approximating the unstable eigenvector

Coefficent	*Value*
T_0	0.0
T_2	.997174015475420839
T_4	-.750076391618631350e-1
T_6	-.421637581906496219e-2
T_8	.243085635200398229e-3
T_{10}	-.350743859605246809e-5
T_{12}	-.116396022929285848e-6
T_{14}	.587775900087177543e-8
T_{16}	-.143726580692062686e-10
T_{18}	-.414960761935413242e-11
T_{20}	.739956444640565298e-13
T_{22}	.187722716176459986e-14
T_{24}	-.984882779975719462e-16
T_{26}	.923369015252735332e-18
T_{28}	.486949561737800715e-19
T_{30}	-.182115607952668435e-20
T_{32}	.114663532949642848e-22
T_{34}	.876584832754696906e-24
T_{36}	-.264316258499150467e-25
T_{38}	.454644401771555384e-28
T_{40}	.145567572718601708e-28
T_{42}	-.315715312149449489e-30
T_{44}	-.246698864030537125e-32
T_{46}	.234890047421995642e-33
T_{48}	-.296813572483154104e-35
T_{50}	-.891377391603921019e-37

Table A.2: Approximation of unstable eigenvector

A.3 Verifying the Estimating Lemma

f_0		
A	$=$	0.00000000036867817673042322252197864089137691046005
B	$=$	0.01566312850568283079888409758950729155912995338440
C	$=$	0.01317824185155894167742474110127659514546394348145
E	$=$	0.48532992178594008825243122373649384826421737670898
M_1	$=$	0.00043607147106710758251504889138061571429716423154
M_2	$=$	0.00870470390406178837205963816359144402667880058289
M_3	$=$	0.00086598869896154753895978206656991460477001965046
ϵ	$=$	0.52570818396938379102323324332246556878089904785156
δ	$=$	0.00000000000041382436273933801740577906697290383990
$E-(A+B+C)$	$=$	0.45648855106002012016475077871291432529687881469727
f_{100} A	$=$	0.00000000011191106244062630099359845514851849110349
B	$=$	0.04389052321926437127963538387120934203267097473145
C	$=$	0.00135724974161681053927419604576698475284501910210
E	$=$	0.60731014572362651549042311671655625104904174804688
M_1	$=$	0.00045941349877648825173956370626626721787033602595
M_2	$=$	0.01105513634795451775871821809005268733017146587372
M_3	$=$	0.00091607134092926275834023464028632588451728224754
ϵ	$=$	5.39957422920855645287474544602446258068084716796875
δ	$=$	0.00000000000040102978521364848968504220990212928071
$E-(A+B+C)$	$=$	0.56206237265083425214129420055542141199111938476562
f_{200}		
A	$=$	0.00000000005139830507026575749174755281743461401978
B	$=$	0.09764410110497429018394655031443107873201370239258
C	$=$	0.00055162992568368041101933840764104388654232025146
E	$=$	0.72919645450706982447997006602236069738864898681641
M_1	$=$	0.00047821267841080951181698122098850944894365966320
M_2	$=$	0.01339207970730051620866785100361084914766252040863
M_3	$=$	0.00096436677249625514431879969023952980933245271444
ϵ	$=$	13.98570639620083788656756951240822672843933105468750
δ	$=$	0.00000000000040727582319473841343047345908888227456
$E-(A+B+C)$	$=$	0.63100072342501345445953120361082255840301513671875
f_{300}		
A	$=$	0.0000000002806900920429508980210972817251178666395
B	$=$	0.17490482000583668087401179036532994359731674194336
C	$=$	0.00029604981599442540141781110563101719890255481005
E	$=$	0.83534012607255014870588638586923480033874511718750
M_1	$=$	0.00048508109937083632038148928167231588304275646806
M_2	$=$	0.01461157608981923303193006802302988944575190544128
M_3	$=$	0.00098291385570183184734738279786370185320265591145

Table A.3, Summary of Estimating Lemma values

ϵ	$=$	$2.65607692381160624961466965032741427421569824218750e1$
δ	$=$	$0.00000000000040035124154764042059627469027423640291$
$E-(A+B+C)$	$=$	$0.66013925622265001713628862489713355898857116699219$
f_{400}		
A	$=$	$0.00000000001766832094596666463538956485077480607765$
B	$=$	$0.27036894148282414329642620032245758920907974243164$
C	$=$	$0.00018610577669922557316091460410945046533015556633$
E	$=$	$0.90616398650730856090262932411860674619674682617188$
M_1	$=$	$0.00049448356165162016609415651657855050871148705482$
M_2	$=$	$0.01512360951800087449647946868935832753777503967285$
M_3	$=$	$0.00100812199464900948374379563432512441067956387997$
ϵ	$=$	$4.33354412755616351660137297585606575012207031250000e1$
δ	$=$	$0.00000000000040105812257494139845737436798035438517$
$E-(A+B+C)$	$=$	$0.63560893923011674289114125713240355253219604492188$
f_{500}		
A	$=$	$0.00000000001204321638924278593122413379504716843087$
B	$=$	$0.36483866242367946064462103095138445496559143066406$
C	$=$	$0.00012614582585723443865395598795231535405036993325$
E	$=$	$0.92743805828689085402061209606472402811050415039062$
M_1	$=$	$0.00050368299176893692025575033355266896251123398542$
M_2	$=$	$0.01411306495212696909824501290131593123078346252441$
M_3	$=$	$0.00101657335411211419660959531796606825082562863827$
ϵ	$=$	$6.44697260304195225444345851428806781768798828125000e1$
δ	$=$	$0.00000000000040115398313893679103704276803899575970$
$E-(A+B+C)$	$=$	$0.56247325002531090021307136339601129293441772460938$
f_{600}		
A	$=$	$0.00000000000629961767285720320478351547601958090351$
B	$=$	$0.43749375736655915503092728613410145044326782226562$
C	$=$	$0.00008936250067327959850736057711984017259965185076$
E	$=$	$0.89914596805515867306013433335465379059314727783203$
M_1	$=$	$0.00051481310196781085528755506075526682252530008554$
M_2	$=$	$0.01183975868918901025450551856010861229151487350464$
M_3	$=$	$0.00100624495856171925455002646998536874889396131039$
ϵ	$=$	$9.00820770216067074898091959767043590545654296875000e1$
δ	$=$	$0.00000000000029075196316986631920637273128858808400$
$E-(A+B+C)$	$=$	$0.46156284818162657268558746181952301412820816040039$
f_{700}		
A	$=$	$0.00000000000486259858246393408704696618835123741748$
B	$=$	$0.46601971907534239791814911768597085028886795043945$
C	$=$	$0.00006509385542146887027471074960160990485746879131$
E	$=$	$0.83590850156656082425854492612415924668312072753906$
M_1	$=$	$0.00052551991471893526146341724114563476177863776684$

Table A.3, Summary of Estimating Lemma values

M_2	=	0.00870201620856096182132777272499879472889006137848
M_3	=	0.00097846608783715214967269258039550550165586173534
ϵ	=	1.20252958624348480043408926576375961303710937500000e2
δ	=	0.00000000000029796766331156874499726462125580944641
$E-(A+B+C)$	=	0.36982368863093434407929294138739351183176040649414
f_{800}		
A	=	0.000000000000373310185542968076361818603967937491798
B	=	0.43480919333029560691628034874156583100557327270508
C	=	0.00004861048670044709903680091311883870730525813997
E	=	0.76126432211386585446888375372509472072124481201172
M_1	=	0.00053397820931646341590520732367508571769576519728
M_2	=	0.00550636960714539699934810812465002527460455894470
M_3	=	0.00094197157869783585109474399743589856370817869902
ϵ	=	1.55023599661127775561908492818474769592285156250000e2
δ	=	0.00000000000029383261130960486981238793649808130955
$E-(A+B+C)$	=	0.32640651829313666043574926334258634597063064575195
f_{900}		
A	=	0.000000000000298667145308390367014015674229984183925
B	=	0.34956750115612061913239472232817206531763076782227
C	=	0.00003745911116708813690589705003297638086223741993
E	=	0.70157706972536404421703082334715873003005981445312
M_1	=	0.00053935034076803950973955092962341950624249875546
M_2	=	0.00292504005226847064655149033285397308645769953728
M_3	=	0.00091021183445118389494787880522608247702009975910
ϵ	=	1.94390482014619323081205948255956172943115234375000e2
δ	=	0.00000000000029407848287973924980534749157398620006
$E-(A+B+C)$	=	0.35197210945508961010830262239323928952217102050781
f_{999}		
A	=	0.000000000000234708521810365621048137667907007764987
B	=	0.30619148926045725023215027249534614384174346923828
C	=	0.00003012517509079290506249573788899454029888147488
E	=	0.67888972182176465786085373110836371779441833496094
M_1	=	0.00054118105933289879089231888542599335778504610062
M_2	=	0.00167239244482921228622529508101024475763551890850
M_3	=	0.00089733181724391947436253458292299001186620444059
ϵ	=	2.38294201322180981605924898758530616760253906250000e2
δ	=	0.00000000000028288496009938245669910870945324275652
$E-(A+B+C)$	=	0.37266810738386946244560249397181905806064605712891

Table A.3: Summary of Estimating Lemma values

A.4 Lanford's contracting eigenvalues

Coefficent	*Value*
0	4.6692016091029906718532038204662016172581855774757686327456e+00
1	1.5962844038269977003948083979473903746182515535626382805321e-01
2	-1.2365271255268702433539365258551661039013200810927047057479e-01
3	-5.7307021066681846884365032258640246804649020688568791082330e-02
4	2.5481238979013134705981243720919742133531108608778125406148e-02
5	-1.0145805672088501245799896475915336061703022830884391006354e-02
6	4.0675304372387237302150267972901306880155689177454733755168e-03
7	-1.6252781653666029690025865257927828818333451407788584632443e-03
8	6.4929353990557833960505411640929947568669720785055388454329e-04
9	-2.5877724716639396988879511746983123718009383714068352867613e-04
10	1.0364571512568970605893031123060211458911749960055082442470e-04
11	-4.1311180104466435415053787259431781272777046885561416741097e-05
12	1.6544803857863443046424585124864186741117787913622306201493e-05
13	-6.5981592561773728451075112665095259904352065699387444095298e-06
14	2.6410212362684157782822591969776860987597675821135995331914e-06
15	-1.0538680067652696892568628130957031627142651985801966209436e-06
16	4.2158210096311685174789102414116531737863700063507516891281e-07
17	-1.6830116908500628887491163165356049213717628286189765197623e-07
18	6.7296493270004213673573407611125437309202837663909785744950e-08
19	-2.6873666972416140401655502595597567003700368071545659760516e-08
20	1.0742434263915623921024353422867111594419289433705765923848e-08

Table A.4: Lanford's approximations of the contracting eigenvectors

Appendix B

The Code

Here follows a selection of the C/C++ code used in the proof. Extensive literature exists on constructing numerical algorithms. We are particularly indebted to [PFTV]. Recall that two different numerical packages are used: CLN and CXSC. We provide a brief overview of some of these packages' key features to enable readability of the code.

B.1 CLN

CLN stands for Class Library for Numbers. It enables efficient computation on a rich set of number classes with arbitrary precision. The package was originally written by Bruno Haible. For more information see http://www.ginac.de/CLN.

The following ordinary number types appear repeatedly in the code:

- `cl_N`: a complex number
- `cl_R`: a real number

both of these are treated as floating-point numbers with a mantissa and an exponent. CLN uses a "round-to-even" rule: first it computes the exact mathematical result and then returns the floating point number nearest to it. If two such numbers are equidistant the one with 0 in its least significant mantissa bit is chosen. We use "Long Floats" (officially type `cl_LF`). They have 1 sign bit, 32 exponent bits (including the exponent's sign), and n mantissa bits (including the "hidden" bit), where $n \geq 64$. The precision of a long float is unlimited, but once created, a long float has a fixed precision.

When specifying constants, the number of significant digits desired can be specified using an underscore. Thus `cl_R One = "1.0_50"` creates a variable called "One" which will be kept to fifty decimal places.

B.2 CXSC

CXSC is a Class library for eXtended Scientific Computing. It's predefined data types include `interval` and `complex interval` which, as we have argued, are necessary for any serious mathematical computing. More information can be found at http://www.math.uni-wuppertal.de/ xsc.

When creating ("casting") to a CXSC type, an underscore is used in front of the type name. So, for example, `interval One = _interval(1.0)` creates the smallest floating point interval guaranteed to contain 1.0.

B.3 Constants

Throughout, we adopt the following constants. Note that the constant real numbers Zero, One, etc., are specified in this manner to assure enough memory allocation for floating point accuracy.

```
cl_R Zero="0.0_50";
cl_R One="1.0_50";
cl_R MOne="-1.0_50";
cl_R Ten="10.0_50";
cl_R Two="2.0_50";
cl_R Four="4.0_50";
cl_R Half="0.5_50";
cl_R Feig="-1.4011552_50";
cl_R FeigDelta="4.669_50";
cl_R Tolerance="1.0e-50_50";

cl_N cZero=cln::complex(Zero,Zero);
cl_N cOne=cln::complex(One,Zero);
cl_N cMOne=cln::complex(MOne,Zero);
cl_N cTen=cln::complex(Ten,Zero);
cl_N cTwo=cln::complex(Two,Zero);
cl_N cFour=cln::complex(Four,Zero);
cl_N cHalf=cln::complex(Half,Zero);
cl_N I=cln::complex(Zero,One);
cl_N cEval=cln::complex("4.66920160910299308_50",Zero);

cl_N Radius=cln::complex("0.0003_50",Zero);
cl_N Per2radius=cln::complex("0.01_50",Zero);
interval iPer2radius=14.0/_interval(1000.0);

const int Deg=25;
const int cDeg=10;
const int Renorms=4;
const int SignificantDigits=14;
const int DerivBoundAcc=10;
const int SquareBoundAcc=10;
```

Note that Deg will give the number of terms in our polynomial approximations. Since all our functions are even, Deg=25 implies our polynomials will be degree 50.

General Note: In a (somewhat unsuccessful) attempt to limit what follows, we have omitted a great deal, particularly with reference to `#define` and `#include` statements. We hope that the effort to increase readability in no way detracts from the executability of the code.

B.4 Classes

We create several new classes to satisfy our needs. Many of these will need to be produced for both CLN and CXSC.

B.4.1 CLN: class Cheby

The `cheby` class represents our (even) Chebyshev approximations (there is a corresponding `chebyOdd` to handle those cases where our approximations are not of even functions).

```
template <class T, int n>
class cheby {

 private:

  T coeff[n+1];

 public:

  //constructors

  cheby();
  cheby( T x );
  cheby( cheby<T,n>& f );

  //operators

  T& operator[] ( int i );

  void operator= ( cheby<T,n>& f);

  cheby<T,n>& operator+ ( cheby<T,n>& f);
  cheby<T,n>& operator+ ( T x[]);
  cheby<T,n>& operator+ ( T x);
  cheby<T,n>& operator- ( cheby<T,n>& f);
  cheby<T,n>& operator- ( T x[]);
  cheby<T,n>& operator- ( T x);

  cheby<T,n>& operator* ( cheby<T,n>& f);
  cheby<T,n>& operator* ( T x[]);
  cheby<T,n>& operator* ( T x);

  cheby<T,n>& operator/ ( cheby<T,n>& f);
  cheby<T,n>& operator/ ( T x[]);
  cheby<T,n>& operator/ ( T x);

  T operator() ( T x );
#ifdef _COMPLEX_
  cl_N operator() ( cl_R y );
#endif

  //friends

  friend int deg<> ( cheby<T,n>& f );
  friend void print<> ( cheby<T,n>& f );
  friend void setZeroCoeff<>( cheby<T,n>& f );
  friend T geta<>( cheby<T,n>& f );
  friend void R<> ( cheby<T,n>& f );
  friend void R<> ( cheby<T,n>& f, int m );

};    // *******************    END OF CLASS   ********************

template <class T, int n>
class chebyOdd {

 private:

  T coeff[n+1];

 public:

  //constructors

  chebyOdd();
  chebyOdd( T x );
  chebyOdd( chebyOdd<T,n>& f );
  //operators

  T& operator[] ( int i );

  void operator= ( chebyOdd<T,n>& f);

  chebyOdd<T,n>& operator+ ( chebyOdd<T,n>& f);
  chebyOdd<T,n>& operator+ ( T x[]);
  chebyOdd<T,n>& operator+ ( T x);

  chebyOdd<T,n>& operator- ( chebyOdd<T,n>& f);
  chebyOdd<T,n>& operator- ( T x[]);
  chebyOdd<T,n>& operator- ( T x);

  chebyOdd<T,n>& operator* ( chebyOdd<T,n>& f);
  chebyOdd<T,n>& operator* ( T x[]);
  chebyOdd<T,n>& operator* ( T x);

  chebyOdd<T,n>& operator/ ( chebyOdd<T,n>& f);
  chebyOdd<T,n>& operator/ ( T x[]);
  chebyOdd<T,n>& operator/ ( T x);

  T operator() ( T x );
#ifdef _COMPLEX_
  cl_N operator() ( cl_R y );
#endif

  //friends

  friend int deg<> ( chebyOdd<T,n>& f );
  friend void print<> ( chebyOdd<T,n>& f );
  friend void D<> ( chebyOdd<T,n>& df, cheby<T,n>& f );
  friend void D<> ( cheby<T,n>& df, chebyOdd<T,n>& f );
};// *******************    END OF CLASS   ********************

// template independent routines

// cheby

//constructors
```

```
template <class T, int n>
inline cheby<T,n>::cheby( T x ) {
  for (int i=0;i<n+1; i++ ) coeff[i]=x;
}

template <class T, int n>
inline cheby<T,n>::cheby( cheby<T,n>& f ) {
  for (int i=0; i<n+1  ; i++ ) coeff[i]=f[i];
}

//operators

template <class T, int n>
inline T& cheby<T,n>::operator[] ( int i ) {
  return coeff[i];
}

template <class T, int n>
inline void cheby<T,n>::operator= ( cheby<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) coeff[i]=f[i];
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator+ ( cheby<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+f[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator+ ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+x[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator+ ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+x;
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator- ( cheby<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-f[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator- ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-x[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator- ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-x;
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator* ( cheby<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*f[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator* ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*x[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator* ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*x;
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator/ ( cheby<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/f[i];
  return *this;
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator/ ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/x[i];
  return *this;
```

```
}

template <class T, int n>
inline cheby<T,n>& cheby<T,n>::operator/ ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/x;
  return *this;
}

//friends

template <class T, int n>
inline int deg ( cheby<T,n>& f ) {
  return 2*n;
}

template <class T, int n>
inline void print ( cheby<T,n>& f ) {
  for (int i=0; i<n+1 ; i++)
    cout << "T["<< 2*i << "]=" << f[i] << endl;
}

template <class T, int n>
inline T geta( cheby<T,n>& f ) {
  return f(One);
}

template <class T, int n>
inline void R ( cheby<T,n>& f, int m ) {
  for (int i=0; i<m; i++) R(f);
}

// chebyOdd

//constructors
template <class T, int n>
inline chebyOdd<T,n>::chebyOdd( T x ) {
  for (int i=0;i<n+1; i++ ) coeff[i]=x;
}

template <class T, int n>
inline chebyOdd<T,n>::chebyOdd( chebyOdd<T,n>& f ) {
  for (int i=0; i<n+1  ; i++ ) coeff[i]=f[i];
}
```

```
//operators

template <class T, int n>
inline T& chebyOdd<T,n>::operator[] ( int i ) {
  return coeff[i];
}

template <class T, int n>
inline void chebyOdd<T,n>::operator= ( chebyOdd<T,n>& f) {
  for ( int i=0; i<n+1 ; i++ ) coeff[i]=f[i];
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator+ ( chebyOdd<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+f[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator+ ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+x[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator+ ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]+x;
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator- ( chebyOdd<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-f[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator- ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-x[i];
  return *this;
}
```

```
template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator- ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]-x;
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator* ( chebyOdd<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*f[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator* ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*x[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator* ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]*x;
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator/ ( chebyOdd<T,n>& f) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/f[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator/ ( T x[]) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/x[i];
  return *this;
}

template <class T, int n>
inline chebyOdd<T,n>& chebyOdd<T,n>::operator/ ( T x) {
  for ( int i=0; i<n+1; i++ ) this->coeff[i]=this->coeff[i]/x;
  return *this;
}

//friends
```

```
template <class T, int n>
inline int deg ( chebyOdd<T,n>& f ) {
  return 2*n-1;
}

template <class T, int n>
inline void print ( chebyOdd<T,n>& f ) {
  cout << "T[0]=" << f[0] << endl;
  for (int i=1; i<n+1 ; i++)
    cout << "T["<< 2*i-1 << "]=" << f[i] << endl;
}

#define _CL_CHEBY_N_INL_
// complex routines

// cheby

//constructors

template <class T, int n>
inline cheby<T,n>::cheby() {
  for (int i=0; i<n+1 ; i++ ) coeff[i]=cZero;
}

//operators

template <class T, int n>
inline T cheby<T,n>::operator() ( T x ) {
  // Evaluate even chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  T d=cZero, dd=cZero, sv=cZero;

  for (int i=n; i > 0; i-- ) {
    sv=d;                           // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];      // Clenshaw's recurrence
    dd=sv;                          // think dd = d(i+2)
    sv=d;
    d=cTwo*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];       // again Clenshaw
```

```
}

template <class T, int n>
inline cl_N cheby<T,n>::operator() ( cl_R y ) {
  // Evaluate even chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cl_N x=cln::complex(y,Zero);
  T d=cZero, dd=cZero, sv=cZero;

  for (int i=n; i > 0; i-- ) {
    sv=d;                               // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];          // Clenshaw's recurrence
    dd=sv;                              // think dd = d(i+2)
    sv=d;
    d=cTwo*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];           // again Clenshaw
}

//friends

template <class T, int n>
inline void setZeroCoeff( cheby<T,n>& f ) {
  // Use the fact we know the form of individual T[i]s to compute c[0]
  // given that f(0)=1.
  T hack=cMOne;
  T sum=cZero;

  for (int i=1; i<n+1; i++) {          // even function only has even terms
    sum=sum+hack*f[i];
    hack=hack*MOne;
  }
  f[0]=cOne - sum;                      // since 1=f(0)=sum+c[0]
}

template <class T, int n>
inline void R ( cheby<T,n>& f ) {
  // given an "arbitrary" function formed by composing a cheby g,
  // a linear function y=mx, and its inverse  turn it into a linear
  // combination of Chebyshev polynomials f. This algorithm stolen in part
  // from Numerical Recipes, p161 -- we can make some simplifications since
  // we are already in the interval [-1,1]. See esp formula on pg 159 since
  // the book seems to have it all wrong.
  T a=geta(f);
  cheby<T,n> g(f);                      // g is a copy of f
  T sum, x;
  T Pi=cln::complex(pi(precision),Zero);
  T N=CAST(2*n+1);                      // Set N.

  for (int j=1; j<n+1; j++) {           // for the jth coeff
    int k=2*j;
    sum = cZero;                         // ah - don't forget to reset sum
    for (int i=1; i <=2*n+1; i++) {   // calculate summation over i
      x = cos( Pi*(CAST(i)-cHalf)/N );
      // x = f(f(a*x))/a;
      x=a*x;
      x=f(x);
      x=f(x);
      x=x/a;
      sum = sum + x * cos( Pi*CAST(k)*(CAST(i) - cHalf)/N );
    }
    g[j]=(cTwo/N)*sum;                   // see formula p159.
  }
  f=g;                                  // now make f=g
  setZeroCoeff(f);
}

// chebyOdd

//constructors
template <class T, int n>
inline chebyOdd<T,n>::chebyOdd() {
  for (int i=0; i<n+1 ; i++ ) coeff[i]=cZero;
}

//operators

template <class T, int n>
inline T chebyOdd<T,n>::operator() ( T x ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  T d=cZero, dd=cZero, sv=cZero;
  for (int i=n; i > 1; i-- ) {
    sv=d;                            // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];       // Clenshaw's recurrence
    dd=sv;                           // think dd = d(i+2)
    sv=d;
    d=cTwo*x*d-dd;
    dd=sv;
  }
  sv=d;
  d=cTwo*x*d - dd + coeff[1];       // do i=1
  dd=sv;
  return x*d - dd + coeff[0];        // again Clenshaw
}

template <class T, int n>
inline cl_N chebyOdd<T,n>::operator() ( cl_R y ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cl_N x=cln::complex(y,Zero);
  T d=cZero, dd=cZero, sv=cZero;

  for (int i=n; i > 1; i-- ) {
    sv=d;                            // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];       // Clenshaw's recurrence
    dd=sv;                           // think dd = d(i+2)
    sv=d;
    d=cTwo*x*d-dd;
    dd=sv;
  }
  sv=d;
  d=cTwo*x*d - dd + coeff[1];       // do i=1
  dd=sv;
  return x*d - dd + coeff[0];        // again Clenshaw
}

//friends

template <class T, int n>
inline void D ( chebyOdd<T,n>& df, cheby<T,n>& f ) {
  // compute deriv. of f
  for ( int i=1; i<n+1; i++ )
    for ( int j=i; j<n+1; j++ )
```

```
      df[i]=df[i]+Four*CAST(j)*f[j];
  df[0]=cZero;
}

template <class T, int n>
inline void D ( cheby<T,n>& df, chebyOdd<T,n>& f ) {
  // compute deriv. of f
  T constant=f[1];
  for ( int i=1; i<n; i++ ) {
    for ( int j=i; j<n; j++ )
      df[i]=df[i]+CAST(4*j+2)*f[j+1];
    constant=constant+CAST(2*i+1)*f[i+1];
  }
  df[0]=constant;
  df[n]=cZero;
}
```

The real incarnation of cheby is similar and the details are omitted.

B.4.2 CXSC: class Cheby

Similar clases must be implemented with interval arithmetic: ccheby (complex), cicheby (complex interval), and cichebyOdd (complex interval for odd functions).

```
class ccheby {

 private:

  cvector coeff;

 public:

  cinterval Tzero;

  //constructors
  ccheby();
  ccheby( int n );
  ccheby( int n, cxsc::complex x );
  ccheby( ccheby& f );

  //operators
  cxsc::complex& operator[] ( int i );

  void operator= ( ccheby& f );

  cinterval operator() ( cinterval x );
  cinterval operator() ( interval x );
  cinterval operator() ( cxsc::complex x );
  cinterval operator() ( real x );

  //friend

  friend int Lb( ccheby& f );
  friend int Ub( ccheby& f );
  friend void Resize (ccheby& f, int lb, int ub);
};

int deg ( ccheby& f );
void print ( ccheby& f );
cinterval getZeroCoeff( ccheby& f );
cinterval geta( ccheby& f );


    // *******************    END OF CLASS   ********************

class cicheby {

 private:

  civector coeff;

 public:

  //constructors
  cicheby();
  cicheby( int n );
  cicheby( int n, cinterval x );
  cicheby( cicheby& f );

  //operators
  cinterval& operator[] ( int i );
```

```
    void operator= ( cicheby& f );

    cicheby& operator+ ( cicheby& f );
    cicheby& operator+ ( civector x );
    cicheby& operator+ ( cvector x );
    cicheby& operator+ ( cinterval x );
    cicheby& operator+ ( cxsc::complex x );

    cicheby& operator- ( cicheby& f );
    cicheby& operator- ( civector x );
    cicheby& operator- ( cvector x );
    cicheby& operator- ( cinterval x );
    cicheby& operator- ( cxsc::complex x );

    cicheby& operator* ( cicheby& f );
    cicheby& operator* ( civector x );
    cicheby& operator* ( cvector x );
    cicheby& operator* ( cinterval x );
    cicheby& operator* ( cxsc::complex x );

    cicheby& operator/ ( cicheby& f );
    cicheby& operator/ ( civector x );
    cicheby& operator/ ( cvector x );
    cicheby& operator/ ( cinterval x );
    cicheby& operator/ ( cxsc::complex x );

    cinterval operator() ( cinterval x );
    cinterval operator() ( interval x );
    cinterval operator() ( cxsc::complex x );
    cinterval operator() ( real x );

    //friend

    friend int Lb( cicheby& f );
    friend int Ub(cicheby& f );
    friend void Resize (cicheby& f, int lb, int ub);

};

int deg ( cicheby& f );
void print ( cicheby& f );
void setZeroCoeff( cicheby& f );
```

```
cinterval geta( cicheby& f );
void R ( cicheby& f );
void Ralt ( cicheby& f );
void R ( cicheby& f, int m );

    // ******************    END OF CLASS   *******************

class cichebyOdd {

 private:

  civector coeff;

 public:

  //constructors

  cichebyOdd();
  cichebyOdd( int n );
  cichebyOdd( int n, cinterval x );
  cichebyOdd( cichebyOdd& f );

  //operators

  cinterval& operator[] ( int i );

  void operator= ( cichebyOdd& f );

  cichebyOdd& operator+ ( cichebyOdd& f );
  cichebyOdd& operator+ ( civector x );
  cichebyOdd& operator+ ( cvector x );
  cichebyOdd& operator+ ( cinterval x );
  cichebyOdd& operator+ ( cxsc::complex x );

  cichebyOdd& operator- ( cichebyOdd& f );
  cichebyOdd& operator- ( civector x );
  cichebyOdd& operator- ( cvector x );
  cichebyOdd& operator- ( cinterval x );
  cichebyOdd& operator- ( cxsc::complex x );

  cichebyOdd& operator* ( cichebyOdd& f );
  cichebyOdd& operator* ( civector x );
  cichebyOdd& operator* ( cvector x );
  cichebyOdd& operator* ( cinterval x );
  cichebyOdd& operator* ( cxsc::complex x );

  cichebyOdd& operator/ ( cichebyOdd& f );
  cichebyOdd& operator/ ( civector x );
  cichebyOdd& operator/ ( cvector x );
  cichebyOdd& operator/ ( cinterval x );
  cichebyOdd& operator/ ( cxsc::complex x );

  cinterval operator() ( cinterval x );
  cinterval operator() ( interval x );
  cinterval operator() ( cxsc::complex x );
  cinterval operator() ( real x );

  //friend

  friend int Lb( cichebyOdd& f );
  friend int Ub(cichebyOdd& f );
  friend void Resize (cichebyOdd& f, int lb, int ub);
};

int deg ( cichebyOdd& f );
void print ( cichebyOdd& f );
void D( cichebyOdd& df, cicheby& f );
void D( cicheby& df, cichebyOdd& f );
void D( cichebyOdd& df, ccheby& f );

   // ******************   END OF CLASS   *******************

  // the inline code ...

inline int min (int m, int n) {
  if ( m < n  ) return m;
  else return n;
}

inline int max (int m, int n) {
  if ( m < n  ) return n;
  else return m;
}

// ccheby class
```

```
inline ccheby::ccheby() {
  Resize(coeff,1,Deg);
  for (int i=1;i<Deg+1; i++ ) coeff[i]=( 0.0,0.0 );
  Tzero=_cinterval(0.0);
}

inline ccheby::ccheby( int n ) {
  Resize(coeff,1,n);
  for (int i=1;i<n+1; i++ ) coeff[i]=( 0.0,0.0 );
  Tzero=_cinterval(0.0);
}

inline ccheby::ccheby( int n, cxsc::complex x ) {
  Resize(coeff,1,n);
  for (int i=1;i<n+1; i++ ) coeff[i]=x;
  Tzero=getZeroCoeff(*this);
}

inline ccheby::ccheby( ccheby& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for (int i=Lb(f); i<=Ub(f)  ; i++ ) coeff[i]=f[i];
  Tzero=f.Tzero;
}

//operators
inline cxsc::complex& ccheby::operator[] ( int i ) {
  return coeff[i];
}

inline void ccheby::operator= ( ccheby& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for ( int i=Lb(f); i<=Ub(f); i++ ) coeff[i]=f[i];
  Tzero=f.Tzero;
}

inline cinterval ccheby::operator() ( cinterval x ) {
  // Evaluate even cchebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                                       // think d = d(i+1)
    d=2.0*x*d - dd + _cinterval(coeff[i]); // Clenshaw's recurrence
    dd=sv;                                  // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + Tzero;                  // again Clenshaw
}

inline cinterval ccheby::operator() ( interval x ) {
  // Evaluate even cchebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                                   // think d = d(i+1)
    d=2.0*x*d - dd + _cinterval(coeff[i]); // Clenshaw's recurrence
    dd=sv;                                  // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + Tzero;                  // again Clenshaw
}

inline cinterval ccheby::operator() ( cxsc::complex y ) {
  // Evaluate even cchebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x=_cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                                   // think d = d(i+1)
    d=2.0*x*d - dd + _cinterval(coeff[i]); // Clenshaw's recurrence
    dd=sv;                                  // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + Tzero;                  // again Clenshaw
}
```

```
inline cinterval ccheby::operator() ( real y ) {
  // Evaluate even cchebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x= _cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                                   // think d = d(i+1)
    d=2.0*x*d - dd + _cinterval(coeff[i]); // Clenshaw's recurrence
    dd=sv;                                  // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + Tzero;                  // again Clenshaw
}

//friends

inline int Lb( ccheby& f ) {
  return Lb(f.coeff);
}

inline int Ub(ccheby& f ) {
  return Ub(f.coeff);
}

inline void Resize (ccheby& f, int lb, int ub) {
  Resize(f.coeff,lb,ub);
}

inline int deg ( ccheby& f ) {
  return 2*Ub(f);
}

inline void print ( ccheby& f ) {

  cout << SetPrecision(25,25) << Scientific;
  cout << "T[0]=" << f.Tzero << endl;
  for (int i=Lb(f); i<=Ub(f) ; i++)
    cout << "T["<< 2*i << "]=" << f[i] << endl;
}

inline cinterval getZeroCoeff( ccheby& f ) {
  // Use the fact we know the form of individual T[i]s to compute c[0]
  // given that f(0)=1.
  int hack=-1;
  cxsc::complex One( 1.0,0.0 );
  cdotprecision accu(0.0);
  cinterval ans;;

  for (int i=1; i<Ub(f)+1; i++) {      // even function only has even terms
    accumulate(accu,hack,f[i]);
    hack=-hack;
  }
  accu=One-accu;                       // since 1=f(0)=sum+c[0]
  ans=rnd(accu);
  return ans;
}

inline cinterval geta( ccheby& f ) {
  cxsc::complex x(1.0);
  return f(x);
}

// end of class
// ******************************************************
// cicheby class

inline cicheby::cicheby() {
  Resize(coeff,0,Deg);
  for (int i=0;i<Deg+1; i++ ) coeff[i]=_cinterval(0.0);
}

inline cicheby::cicheby( int n ) {
  Resize(coeff,0,n);
  for (int i=0;i<n+1; i++ ) coeff[i]=_cinterval(0.0);
}

inline cicheby::cicheby( int n, cinterval x ) {
  Resize(coeff,0,n);
  for (int i=0;i<n+1; i++ ) coeff[i]=x;
}
```

```
inline cicheby::cicheby( cicheby& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for (int i=Lb(f); i<=Ub(f)  ; i++ ) coeff[i]=f[i];
}

//operators
inline cinterval& cicheby::operator[] ( int i ) {
  return coeff[i];
}

inline void cicheby::operator= ( cicheby& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for ( int i=Lb(f); i<=Ub(f); i++ ) coeff[i]=f[i];
}

inline cicheby& cicheby::operator+ ( cicheby& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+f[i];
  return *this;
}

inline cicheby& cicheby::operator+ ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+x[i];
  return *this;
}

inline cicheby& cicheby::operator+ ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+_cinterval(x[i]);
  return *this;
}

inline cicheby& cicheby::operator+ ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]+x;
  return *this;
}

inline cicheby& cicheby::operator+ ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]+_cinterval(x);
  return *this;
}
```

```
inline cicheby& cicheby::operator- ( cicheby& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]-f[i];
  return *this;
}

inline cicheby& cicheby::operator- ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]-x[i];
  return *this;
}

inline cicheby& cicheby::operator- ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]-_cinterval(x[i]);
  return *this;
}

inline cicheby& cicheby::operator- ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]-x;
  return *this;
}

inline cicheby& cicheby::operator- ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]-_cinterval(x);
  return *this;
}

inline cicheby& cicheby::operator* ( cicheby& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*f[i];
  return *this;
}

inline cicheby& cicheby::operator* ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*x[i];
  return *this;
}

inline cicheby& cicheby::operator* ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*_cinterval(x[i]);
  return *this;
}

inline cicheby& cicheby::operator* ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]*x;
  return *this;
}

inline cicheby& cicheby::operator* ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]*_cinterval(x);
  return *this;
}

inline cicheby& cicheby::operator/ ( cicheby& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/f[i];
  return *this;
}

inline cicheby& cicheby::operator/ ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/x[i];
  return *this;
}

inline cicheby& cicheby::operator/ ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/_cinterval(x[i]);
  return *this;
}

inline cicheby& cicheby::operator/ ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]/x;
  return *this;
}

inline cicheby& cicheby::operator/ ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]/_cinterval(x);
  return *this;
```

```
}

inline cinterval cicheby::operator() ( cinterval x ) {
  // Evaluate even cichebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                              // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];         // Clenshaw's recurrence
    dd=sv;                             // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];          // again Clenshaw
}

inline cinterval cicheby::operator() ( interval x ) {
  // Evaluate even cichebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                              // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];         // Clenshaw's recurrence
    dd=sv;                             // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];          // again Clenshaw
}

inline cinterval cicheby::operator() ( cxsc::complex y ) {
  // Evaluate even cichebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x=_cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;
  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                               // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];          // Clenshaw's recurrence
    dd=sv;                              // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];           // again Clenshaw
}

inline cinterval cicheby::operator() ( real y ) {
  // Evaluate even cichebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x= _cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>0; i-- ) {
    sv=d;                               // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];          // Clenshaw's recurrence
    dd=sv;                              // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  return x*d - dd + coeff[0];           // again Clenshaw
}

//friends

inline int Lb( cicheby& f ) {
  return Lb(f.coeff);
}

inline int Ub(cicheby& f ) {
  return Ub(f.coeff);
}

inline void Resize (cicheby& f, int lb, int ub) {
  Resize(f.coeff,lb,ub);
}
```

```
inline int deg ( cicheby& f ) {
  return 2*Ub(f);
}

inline void print ( cicheby& f ) {
  cout << SetPrecision(25,25) << Scientific;
  for (int i=Lb(f); i<=Ub(f) ; i++)
    cout << "T["<< 2*i << "]=" << f[i] << endl;
}

inline void setZeroCoeff( cicheby& f ) {
  // Use the fact we know the form of individual T[i]s to compute c[0]
  // given that f(0)=1.
  int hack=-1;
  cinterval sum=_cinterval(0.0);

  for (int i=1; i<Ub(f)+1; i++) {      // even function only has even terms
    sum=sum+hack*f[i];
    hack=-hack;
  }
  f[0]=1.0 - sum;                     // since 1=f(0)=sum+c[0]
}

inline cinterval geta( cicheby& f ) {
  cxsc::complex x(1.0);
  return f(x);
}

inline void Ralt ( cicheby& f ) {
  // given an "arbitrary" function formed by composing a cicheby g,
  // a linear function y=mx, and its inverse  turn it into a linear
  // combination of Cichebyshev polynomials f. This algorithm stolen in part
  // from Numerical Recipes, p161 -- we can make some simplifications since
  // we are already in the interval [-1,1]. See esp formula on pg 159 since
  // the book seems to have it all wrong.

  cinterval Half=5.0/_cinterval(10);

  int n=Ub(f);
  int m=2*n*n;;

  Resize(f,0,m);
  cinterval a=geta(f);
  cicheby g(f);                       // g is a copy of f
  cinterval sum, x;
  interval Pi( acos(_interval(-1.0)) );
  cinterval N=_cinterval(2*m+1);      // Set N.

  for (int j=1; j<m+1; j++) {         // for the jth coeff
    int k=2*j;
    cinterval sum = _cinterval(0.0);  // ah - don't forget to reset sum
    for (int i=1; i <=2*m+1; i++) {   // calculate summation over i
      x = cos( Pi* (i-Half) /N );
      x = f(f(a*x))/a;
      sum=x*cos( Pi*k*(i-Half)/N );
    }
    g[j]=2.0/N*sum;                     // see formula p159.
  }
  f=g;                                // now make f=g
  setZeroCoeff(f);
}


inline void R ( cicheby& f ) {
  // given an "arbitrary" function formed by composing a cheby g,
  // a linear function y=mx, and its inverse  turn it into a linear
  // combination of Chebyshev polynomials f. This algorithm stolen in part
  // from Numerical Recipes, p161 -- we can make some simplifications since
  // we are already in the interval [-1,1]. See esp formula on pg 159 since
  // the book seems to have it all wrong.

  cinterval Half=5.0/_cinterval(10);
  int n=Ub(f);
  cinterval a=geta(f);
  cicheby g(f);                       // g is a copy of f
  cinterval sum, x;
  interval Pi( acos(_interval(-1.0)) );
  cinterval N=_cinterval(2*n+1);      // Set N.

  for (int j=1; j<n+1; j++) {       // for the jth coeff
    int k=2*j;
    sum = _cinterval(0.0); // ah - don't forget to reset sum
    for (int i=1; i <=2*n+1; i++) {  // calculate summation over i
      x = cos( Pi*(i-Half)/N);
      x = a*x;
      x = f(x);
      x = f(x);
      x = x/a;
      sum = sum+x*cos( (Pi*k*(i-Half))/N );
    }
    g[j]=(2.0/N)*sum;                // see formula p159.
  }
  f=g;                               // now make f=g
  setZeroCoeff(f);
}

inline void R ( cicheby& f, int m ) {
  for (int i=0; i<m; i++) R(f);
}

// end of class
// **************************************************
// cichebyOdd class

inline cichebyOdd::cichebyOdd() {
  Resize(coeff,0,Deg);
  for (int i=0;i<Deg+1; i++ ) coeff[i]=_cinterval(0.0);
}

inline cichebyOdd::cichebyOdd( int n ) {
  Resize(coeff,0,n);
  for (int i=0;i<n+1; i++ ) coeff[i]=_cinterval(0.0);
}

inline cichebyOdd::cichebyOdd( int n, cinterval x ) {
  Resize(coeff,0,n);
  for (int i=0;i<n+1; i++ ) coeff[i]=x;
}

inline cichebyOdd::cichebyOdd( cichebyOdd& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for (int i=Lb(f); i<=Ub(f)  ; i++ ) coeff[i]=f[i];
}

//operators
inline cinterval& cichebyOdd::operator[] ( int i ) {
  return coeff[i];
}
```

```
inline void cichebyOdd::operator= ( cichebyOdd& f ) {
  Resize(coeff,Lb(f),Ub(f));
  for ( int i=Lb(f); i<=Ub(f); i++ ) coeff[i]=f[i];
}

inline cichebyOdd& cichebyOdd::operator+ ( cichebyOdd& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+f[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator+ ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+x[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator+ ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]+_cinterval(x[i]);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator+ ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]+x;
  return *this;
}

inline cichebyOdd& cichebyOdd::operator+ ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]+_cinterval(x);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator- ( cichebyOdd& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]-f[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator- ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
```

```
    this->coeff[i]=this->coeff[i]-x[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator- ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]-_cinterval(x[i]);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator- ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]-x;
  return *this;
}

inline cichebyOdd& cichebyOdd::operator- ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]-_cinterval(x);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator* ( cichebyOdd& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*f[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator* ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*x[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator* ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]*_cinterval(x[i]);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator* ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]*x;
  return *this;
}
```

```
inline cichebyOdd& cichebyOdd::operator* ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]*_cinterval(x);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator/ ( cichebyOdd& f ) {
  for ( int i=max(Lb(f),Lb(this->coeff)); i<=min(Ub(f),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/f[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator/ ( civector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/x[i];
  return *this;
}

inline cichebyOdd& cichebyOdd::operator/ ( cvector x ) {
  for ( int i=max(Lb(x),Lb(this->coeff)); i<=min(Ub(x),Ub(this->coeff)); i++ )
    this->coeff[i]=this->coeff[i]/_cinterval(x[i]);
  return *this;
}

inline cichebyOdd& cichebyOdd::operator/ ( cinterval x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ ) this->coeff[i]=this->coeff[i]/x;
  return *this;
}

inline cichebyOdd& cichebyOdd::operator/ ( cxsc::complex x ) {
  for ( int i=Lb(coeff); i<=Ub(coeff); i++ )
    this->coeff[i]=this->coeff[i]/_cinterval(x);
  return *this;
}

inline cinterval cichebyOdd::operator() ( cinterval x ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>1; i-- ) {
    sv=d;                           // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];      // Clenshaw's recurrence
    dd=sv;                          // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  sv=d;                           // think d = d(i+1)
  d=2.0*x*d - dd + coeff[1];      // Clenshaw's recurrence
  dd=sv;                          // think dd = d(i+2)
  return x*d - dd + coeff[0];       // again Clenshaw
}

inline cinterval cichebyOdd::operator() ( interval x ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>1; i-- ) {
    sv=d;                           // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];      // Clenshaw's recurrence
    dd=sv;                          // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  sv=d;                           // think d = d(i+1)
  d=2.0*x*d - dd + coeff[1];      // Clenshaw's recurrence
  dd=sv;                          // think dd = d(i+2)
  return x*d - dd + coeff[0];       // again Clenshaw
}

inline cinterval cichebyOdd::operator() ( cxsc::complex y ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x=_cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>1; i-- ) {
    sv=d;                           // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];      // Clenshaw's recurrence
    dd=sv;                              // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  sv=d;                               // think d = d(i+1)
  d=2.0*x*d - dd + coeff[1];          // Clenshaw's recurrence
  dd=sv;                              // think dd = d(i+2)
  return x*d - dd + coeff[0];           // again Clenshaw
}

inline cinterval cichebyOdd::operator() ( cxsc::real y ) {
  // Evaluate odd chebys at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical Recipes, p162

  cinterval x=_cinterval(y);
  cinterval d=_cinterval(0.0), dd=_cinterval(0.0), sv;

  for (int i=Ub(coeff); i>1; i-- ) {
    sv=d;                               // think d = d(i+1)
    d=2.0*x*d - dd + coeff[i];          // Clenshaw's recurrence
    dd=sv;                              // think dd = d(i+2)
    sv=d;
    d=2.0*x*d-dd;
    dd=sv;
  }
  sv=d;                               // think d = d(i+1)
  d=2.0*x*d - dd + coeff[1];          // Clenshaw's recurrence
  dd=sv;                              // think dd = d(i+2)
  return x*d - dd + coeff[0];           // again Clenshaw
}

//friends

inline int Lb( cichebyOdd& f ) {
  return Lb(f.coeff);
}

inline int Ub(cichebyOdd& f ) {
  return Ub(f.coeff);
}

inline void Resize (cichebyOdd& f, int lb, int ub) {
  Resize(f.coeff,lb,ub);
}

inline int deg ( cichebyOdd& f ) {
  return 2*Ub(f)-1;
}

inline void print ( cichebyOdd& f ) {
  cout << SetPrecision(25,25) << Scientific;
  cout << "T[0]=" << f[0] << endl;
  for (int i=1; i<=Ub(f) ; i++)
    cout << "T["<< 2*i-1 << "]=" << f[i] << endl;
}

inline void D ( cichebyOdd& df, cicheby& f ) {
  // compute deriv. of f
  Resize(df,Lb(f),Ub(f));
  for ( int i=1; i<Ub(f)+1; i++ )
    for ( int j=i; j<Ub(f)+1; j++ )
      df[i]=df[i]+(4*j)*f[j];
  df[0]=_cinterval(0.0);
}

inline void D ( cicheby& df, cichebyOdd& f ) {
  // compute deriv. of f
  Resize(df,Lb(f),Ub(f)-1);
  cinterval constant=f[1];
  for ( int i=1; i<Ub(f); i++ ) {
    for ( int j=i; j<Ub(f); j++ )
      df[i]=df[i]+(2*(2*j+1))*f[j+1];
    constant=constant+(2*i+1)*f[i+1];
  }
  df[0]=constant;
}

inline void D ( cichebyOdd& df, ccheby& f ) {
  // compute deriv. of f
  Resize(df,max(Lb(f)-1,0),Ub(f));
  for ( int i=1; i<Ub(f)+1; i++ )
    for ( int j=i; j<Ub(f)+1; j++ )
      df[i]=df[i]+(4*j)*_cinterval(f[j]);
  df[0]=_cinterval(0.0);
}
```

B.4.3 CLN: class pSeries – The Unstable Manifold

Finally, we need classes to implement the ideas of Section 3. Recall we essentially need double power series (and therefore Chebyshev polynomials of arbitrary degree). These are realized in the classes `pSeries` and `allCheby`.

```
class allCheby {

 private:

  cl_N coeff[cDeg+1];

 public:

  // constructors

  allCheby();
  allCheby(cl_N x);
  allCheby( allCheby& f);
  allCheby( cheby<cl_N,Deg>& f );

  // operators

  cl_N& operator[] (int i);

  void operator=( cl_R x);
  void operator=( cl_N f);
  void operator=( allCheby& f);
  void operator=( cheby<cl_N,Deg>& f);

  cl_N operator() (cl_N x);
  cl_N operator() (cl_R x);

};

void print ( allCheby& f);
```

```
    // ******************    END OF CLASS   *******************

class pSeries {


 private:

  allCheby coeff[Deg+1];

 public:

  // constructors

  pSeries();
  pSeries(cl_N x);

  // operators

  allCheby& operator[] (int i);
  cl_N operator() (cl_N t, cl_N z);
  cl_N operator() (cl_R t, cl_R z);

};

void T(pSeries& w, cheby<cl_N,Deg>& f, cl_N t);
void print(pSeries& w);
void formatPrint(pSeries & w, int iter, cl_R change, char* file);
void inputw(pSeries& w, char* file);
cl_R compare(pSeries& w, pSeries& v);
void print(pSeries& w, cl_N t);
cl_N geta(pSeries& w, cl_N t);
void R(pSeries& w);
cl_N projv(pSeries& w, cl_N t, cl_N unitunstable[], cheby<cl_N,Deg>& fix);
inline cl_N preimR(pSeries& w, cl_N t, int n, cl_N unitunstable[],
                   cheby<cl_N,Deg>& fix);
inline void Rlambda(pSeries& w, int n, cl_N unitunstable[],
                    cheby<cl_N,Deg>& fix);
//void Rlambda(pSeries& w, int n);
void R(pSeries& w, int n);
void lambdaDiv(pSeries& w, cheby<cl_N,Deg>& fix, int n);

    // ******************    END OF CLASS   *******************
// and the inline code ...

// allCheby class

// constructors

inline allCheby::allCheby () {
  for (int i=0; i<cDeg+1; i++ ) coeff[i]=cZero;
}

inline allCheby::allCheby ( cl_N x ) {
  for ( int i=0; i<cDeg+1; i++ ) coeff[i]=x;
}

inline allCheby::allCheby( allCheby& f ) {
  for ( int i=0; i<cDeg+1; i++ ) coeff[i]=f[i];
}

inline allCheby::allCheby( cheby<cl_N,Deg>& f) {
  if ( deg(f) < cDeg ) screenErr("cheby too small for allCheby conversion");
  coeff[0]=f[0];
  for (int i=1; 2*i<cDeg+1; i++ ) {
    coeff[2*i]=f[i];
    coeff[2*i-1]=cZero;
  }
}

// operators

inline cl_N& allCheby::operator[] (int i) {
  return coeff[i];
}

inline void allCheby::operator= (cl_R x) {
  for ( int i=0; i<=cDeg; i++ ) coeff[i]=x;
}

inline void allCheby::operator= (cl_N x) {
  for ( int i=0; i<=cDeg; i++ ) coeff[i]=x;
}

inline void allCheby::operator= (allCheby& f) {
  for ( int i=0; i<=cDeg; i++ ) coeff[i]=f[i];
}

inline void allCheby::operator= (cheby<cl_N,Deg>& f) {
  if ( deg(f) < cDeg ) screenErr("cheby too small for allCheby conversion");
  coeff[0]=f[0];
  for (int i=1; 2*i<cDeg+1; i++ ) {
    coeff[2*i]=f[i];
    coeff[2*i-1]=cZero;
  }
}

inline cl_N allCheby::operator() ( cl_N x) {
  // Evaluate cheby at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical REcipes, p162

  cl_N d=cZero, dd=cZero, sv=cZero;

  for (int i=cDeg; i > 0; i-- ) {
    sv=d;                              // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];         // Clenshaw's recurrence
    dd=sv;                             // think dd = d(i+2)
  }
  return x*d - dd + coeff[0];          // again Clenshaw
}

inline cl_N allCheby::operator() ( cl_R x) {
  // Evaluate cheby at x using Clenshaw's recurrence.
  // Algorithm stolen from Numerical REcipes, p162

  cl_N d=cZero, dd=cZero, sv=cZero;

  for (int i=cDeg; i > 0; i-- ) {
    sv=d;                              // think d = d(i+1)
    d=cTwo*x*d - dd + coeff[i];         // Clenshaw's recurrence
    dd=sv;                             // think dd = d(i+2)
  }
  return x*d - dd + coeff[0];          // again Clenshaw
}

// friends

inline void print (allCheby& f) {
  for ( int i=0; i<cDeg+1; i++ )
    cout << "c[" << i << "]=" << f[i] << endl;
}

// end of allCheby class


// pseries class


// constructors

inline pSeries::pSeries () {
  for ( int i=1; i<Deg+1; i++ ) coeff[i]=cZero;
}

inline pSeries::pSeries ( cl_N x ) {
  for ( int i=1; i<Deg+1; i++ ) coeff[i]=x;
}

// operators

inline allCheby& pSeries::operator[] (int i) {
  return coeff[i];
}

inline cl_N pSeries::operator() ( cl_N t, cl_N z ) {
  // t is param value, z is where resulting function is evaluated

  cheby<cl_N,Deg> w(cZero);
  for ( int i=1; i<min(Deg+1,cDeg+1); i++ ) w[i]=coeff[i](t);
  setZeroCoeff(w);
  return w(z);
}

inline cl_N pSeries::operator() ( cl_R t, cl_R z ) {
  // t is param value, z is where resulting function is evaluated
  cheby<cl_N,Deg> w(cZero);
  for ( int i=1; i<min(Deg+1,cDeg+1); i++ ) w[i]=coeff[i](t);
  setZeroCoeff(w);
  return w(z);
}
```

```
inline void T(pSeries& w, cheby<cl_N,Deg>& f, cl_N t) {
  // sets f=w(t)
  for ( int i=1; i<Deg+1; i++ ) f[i]=w[i](t);
  setZeroCoeff(f);
}

inline void print( pSeries& w ) {
  for (int i=1; i<Deg+1; i++ ) {
    cout << "*** c[" << 2*i << "]=" << endl;
    print(w[i]);
    cout << endl;
  }
}

inline void formatPrint ( pSeries& w, int iter, cl_R change, char* file) {
  char datafile[80]="";
  strcpy(datafile, file);
  strcat(datafile, ".change");

  ofstream fout1(datafile, ios::app);
  if ( !fout1.good() ) {
    string fileErr="Can't open ";
    fileErr.append(file);
    screenErr(fileErr);
  }
  fout1 << "chg=" << change << " @ i " << iter << endl;
  fout1.close();

  //ofstream fout(file, ios::trunc);
  ofstream fout(file, ios::app);
  if ( !fout.good() ) {
    string fileErr="Can't open ";
    fileErr.append(file);
    screenErr(fileErr);
  }
  fout << "chg=" << change << " at iter. " << iter << endl << endl;
  for ( int i=1; i<Deg+1; i++ ) {
    for ( int j=0; j<cDeg+1; j++ )
      fout << "w[" << i << "][" << j << "]=cln::complex(\""
           << realpart(w[i][j]) << "_50\",\""
           << imagpart(w[i][j]) << "_50\");" << endl;
    fout << endl;
  }
  fout.close();

  char datafile2[80]="";
  strcpy(datafile2, file);
  strcat(datafile2, ".current");

  ofstream fout2(datafile2, ios::trunc);
  //ofstream fout(file, ios::app);
  if ( !fout2.good() ) {
    string fileErr="Can't open ";
    fileErr.append(file);
    screenErr(fileErr);
  }
  fout2 << "chg=" << change << " at iter. " << iter << endl << endl;
  for ( int i=1; i<Deg+1; i++ ) {
    for ( int j=0; j<cDeg+1; j++ )
      fout2 << "w[" << i << "][" << j << "]=cln::complex(\""
            << realpart(w[i][j]) << "_50\",\"" << imagpart(w[i][j])
            << "_50\");" << endl;
    fout2 << endl;
  }
  fout2.close();

}

inline void inputw(pSeries& w, char* file) {
  ifstream fin(file);
  if ( !fin.good()) screenErr("Can't open");
  for ( int i=1; i<Deg+1; i++ )
    for ( int j=0; j<cDeg+1; j++ ) {
      cl_R temp;
      fin >> temp;
     w[i][j]=cln::complex(temp,Zero);
    }
  fin.close();
}

inline cl_R compare(pSeries& w, pSeries& v) {
  cl_R ans=Zero;
  for ( int i=1; i<Deg+1; i++ )
    for ( int j=0; j<cDeg+1; j++ )
      ans=ans+abs(w[i][j]-v[i][j]);
  return ans;
}

inline void print( pSeries& w, cl_N t ) {
  cheby<cl_N,Deg> f;
  T(w,f,t);
  print(f);
}

inline cl_N geta(pSeries& w, cl_N t) {
  cheby<cl_N,Deg>f;
  T(w,f,t);
  return geta(f);
}

inline void R(pSeries& w) {
  // given power series w(t,z)=k(t)+sum_i[  c_(2*i)(t) * T_(2*i)(z) ]
  // with c_(2*i)(t) = sum_j [ b_(2*i,j) * T_i(t) ]
  // we want to generate a new power series which represents R(w)
  // this means computing new b guys
  // to do this we have to fix a t. this gives an honest function f
  // whose cheby approximation we can then compute explicitly.
  // the b guys then have to appoximate a function which generates the
  // computed cheby appoximation.

  pSeries v(w);

  cl_N sum,t;
  cl_N Pi=cln::complex(pi(precision),Zero);
  cl_N N=CAST(cDeg+1);                      // Set N.

  for (int k=1; k<Deg+1; k++)               // for c_k

    for (int j=0; j<cDeg+1; j++ ) {         // for b_j
      sum=cZero;                            // reset sum
      for (int i=1; i<=cDeg+1; i++ ) {      // sum, see NR bottom p 159
        t = cos( Pi*(CAST(i)-cHalf)/N );    // NB: THIS IS T-VALUE !!!

        // now have t-value, can generate genuine function f

        cheby<cl_N,Deg> f;
        T(w,f,t);                           // set f=w(t)
        R(f);
        t=f[k];                          // answer is kth cheby coeff of Rf
        sum = sum + t * cos( Pi*CAST(j)*(CAST(i) - cHalf)/N ); // NR, p159
      }
      v[k][j]=(cTwo/N)*sum;              // here is b_j ... see NR, p159
      if ( j==0 ) v[k][0]=v[k][0]/cTwo;    // stoopid constant term issue
    }
  w=v;
}

inline cl_N projv(pSeries& w, cl_N t,
                  cl_N unitunstable[], cheby<cl_N,Deg>& fix) {

  // return ( w(t) - fix ) . unitunstable ... the proj of w(t) on v with 0=fix

  if ( abs(vnorm(unitunstable,0,25)-One) > Tolerance )
    screenErr("Not unit vector in projv!" );

  cheby<cl_N,Deg> f;
  T(w,f,t);                             // set f=w(t)
  f=f-fix;                              // set origin
  return dotprod(f,unitunstable);
}


inline cl_N preimR(pSeries& w, cl_N t, int n, cl_N unitunstable[],
                   cheby<cl_N,Deg>& fix) {

  // return x s.t projv( w(x) ) * eval^n = projv( w(t) )

  cl_N lambda=cOne;
  for (int i=0; i<n; i++ ) lambda=lambda*cEval;

  // use false pos as in getPer2 ... probably a better way to do it

  cl_N x1=cZero;
  cl_N x2=t;                           // initial guess is between 0 and t
  cl_N fl,fh,xl,xh,swap,dx,del,funcVal,ans;

  cl_N projvf=projv(w,x2,unitunstable,fix);
// solving proj(w(x))*lambda-proj(f)=0
  fl=projv(w,x1,unitunstable,fix)*lambda - projvf;
  fh=projvf*lambda - projvf;
  if ( realpart(fl)*realpart(fh) > Zero )
     screenErr("Root not bracketed!");

   if ( realpart(fl) < Zero ) {
     xl=x1;
     xh=x2;
   } else {
     xl=x2;
     xh=x1;
     swap=fl;
     fl=fh;
     fh=swap;
   }

   dx=xh-xl;
   for ( int j=1; j<=200; j++) {
     ans=xl+dx*fl/(fl-fh);
     funcVal=projv(w,ans,unitunstable,fix)*lambda - projvf;
     if ( realpart(funcVal) < Zero ) {
       del=xl-ans;
       xl=ans;
       fl=funcVal;
     } else {
       del=xh-ans;
       xh=ans;
       fh=funcVal;
     }
     dx=xh-xl;
     if ( abs(del) < Tolerance || abs(funcVal) < Tolerance ) {
       return ans;
     }
   }
   screenErr("Maximum number of iterations exceeded in root finding!");
 }

 /*

 inline void Rlambda(pSeries& w, int n) {
   // given power series w(t,z)=k(t)+sum_i[  c_(2*i)(t) * T_(2*i)(z) ]
   // with c_(2*i)(t) = sum_j [ b_(2*i,j) * T_i(t) ]
   // we want to generate a new power series which represents R(w)
   // this means computing new b guys
   // to do this we have to fix a t. this gives an honest function f
  // whose cheby approximation we can then compute explicitly.
  // the b guys then have to appoximate a function which generates the
  // computed cheby appoximation.

  pSeries v(w);

  cl_N sum,t;
  cl_N Pi=cln::complex(pi(precision),Zero);
  cl_N N=CAST(cDeg+1);                        // Set N.

  for (int k=1; k<Deg+1; k++)                 // for c_k

    for (int j=0; j<cDeg+1; j++ ) {           // for b_j
      sum=cZero;                              // reset sum
      for (int i=1; i<=cDeg+1; i++ ) {        // sum, see NR bottom p 159
        t = cos( Pi*(CAST(i)-cHalf)/N );      // NB: THIS IS T-VALUE !!!

        // want to divide t by eval^n

        cl_N lambda=cOne;
        for ( int b=0; b<n; b++ ) lambda=lambda*cEval;
        //t=t*lambda;                         // NOTE: MUST BE TIMES -- IDIOT*2!!!

        // now have t-value, can generate genuine function f

        cheby<cl_N,Deg> f;
        T(w,f,t*lambda);                      // set f=w(t)
        R(f,n);
        cl_N ft=f[k];                         // answer is kth cheby coeff of Rf
        sum = sum + ft * cos( Pi*CAST(j)*(CAST(i) - cHalf)/N ); // NR, p159
      }
      v[k][j]=(cTwo/N)*sum;                   // here is b_j ... see NR, p159
      if ( j==0 ) v[k][0]=v[k][0]/cTwo;       // stoopid constant term issue
    }
  w=v;
}

*/

inline void Rlambda(pSeries& w, int n, cl_N unitunstable[],
                    cheby<cl_N,Deg>& fix) {
  // given power series w(t,z)=k(t)+sum_i[  c_(2*i)(t) * T_(2*i)(z) ]
  // with c_(2*i)(t) = sum_j [ b_(2*i,j) * T_i(t) ]
// we want to generate a new power series which represents R(w)
// this means computing new b guys
// to do this we have to fix a t. this gives an honest function f
// whose cheby approximation we can then compute explicitly.
// the b guys then have to appoximate a function which generates the
// computed cheby appoximation.


// modified 1/20/04 with mult/div issue resolved !!!
// modified again 3/15/04 to change to 0,1 and redo loops

pSeries v(w);

cl_N tzero=preimR(w,cOne,n,unitunstable,fix);

cl_N sum,t;
cl_N Pi=cln::complex(pi(precision),Zero);
cl_N N=CAST(cDeg+1);                    // Set N.

for (int k=1; k<Deg+1; k++)             // for c_k

  for (int j=0; j<cDeg+1; j++ ) {       // for b_j
    sum=cZero;                          // reset sum
    for (int i=1; i<=cDeg+1; i++ ) {    // sum, see NR bottom p 159
      t = cos( Pi*(CAST(i)-cHalf)/N );  // NB: THIS IS T-VALUE !!!

    // now have t-value, can generate genuine function f
    // want to use interval [a,b]=[0,tzero]
    // bma=.5*(b-a)=.5*tzero
    // bpa=.5*(b+a)=.5*tzero
    // t*bma+bpa=t*.5*tzero+.5*tzero=.5*tzero(t+1)

      cheby<cl_N,Deg> f;
      cl_N newt=cHalf*tzero*(t+cOne);
      T(w,f,newt);                      // set f=w(t*tzero)
      R(f,n);
      cl_N ft=f[k];                     // answer is kth cheby coeff of Rf
      sum = sum + ft * cos( Pi*CAST(j)*(CAST(i) - cHalf)/N ); // NR, p159
    }
    v[k][j]=(cTwo/N)*sum;               // here is b_j ... see NR, p159
    if ( j==0 ) v[k][0]=v[k][0]/cTwo;   // stoopid constant term issue
  }
w=v;
```

```
}

inline void R(pSeries& w, int n) {
  for (int i=0; i<n; i++) R(w);
}

inline void lambdaDiv(pSeries& w, cheby<cl_N,Deg>& fix, int n) {
  // compute 1/lambda^n * w
  // this is really (w-fix)/lambda^n + fix

  cl_N lambda=cOne;

  if ( n==0 ) screenErr("choose n>0 for lambdaDiv");
  for ( int i=1; i<Deg+1; i++ ) w[i][0]=w[i][0]-fix[i];
  for ( int i=0; i<n; i++ ) lambda=lambda*cEval;
  for ( int i=1; i<Deg+1; i++ )
    for ( int j=0; j<cDeg+1; j++ ) w[i][j]=w[i][j]/lambda;
  for ( int i=1; i<Deg+1; i++ ) w[i][0]=w[i][0]+fix[i];
  /*
  for ( int i=1; i<Deg+1; i++ ) {
    cl_N parity=-cOne;
    cl_N sum=cZero;
    for ( int j=1; 2*j<cDeg+1; j++ ) sum=sum+CAST(j)*parity*w[i][2*j];
    w[i][0]=fix[i]-sum;
  }
  */
}

// end of pseries class
```

B.5 Routines

B.5.1 Communicating between CLN and CXSC

The following routines enable type conversion and related issues pertaining to communication between CLN and CXSC.

```
#ifdef _COMPLEX_
void rnd(cicheby& cxscF, cheby<cl_N,Deg>& clnF);
void rnd(civector& cxscV, cl_N clnV[]);
#endif
#ifdef _REAL_
void rnd(cicheby& cxscF, cheby<cl_R,Deg>& clnF);
#endif
const interval rnd(cl_R x);
const cinterval rnd(cl_N x);
const real rnd(cl_R x, const cl_R (*func)(cl_R));
const cxsc::complex rnd(cl_N x, const cl_R (*func)(cl_R));
const cl_R up(cl_R x);
const cl_R down(cl_R x);
const cl_R chop(cl_R x, const cl_R (*func)(cl_R));
const real stringConv(cl_R x_cln);

#ifdef _COMPLEX_
void rnd(cicheby& cxscF, cheby<cl_N,Deg>& clnF) {
  Resize(cxscF,0,Deg);
  for ( int j=0; j<=Deg; j++ )
    cxscF[j]=rnd(clnF[j]);
}

void rnd(civector& cxscV, cl_N clnV[]) {
  Resize(cxscV,0,Deg);
  for ( int j=0; j<=Deg; j++ )
    cxscV[j]=rnd(clnV[j]);
}

#endif

#ifdef _REAL_
void rnd(cicheby& cxscF, cheby<cl_R,Deg>& clnF) {
  Resize(cxscF,0,Deg);
  for ( int j=0; j<=Deg; j++ )
    cxscF[j]=( rnd(clnF[j]) , 0.0 );
}
#endif

const interval rnd(cl_R x) {
  interval ans( rnd(x,down), rnd(x,up) );
  return ans;
}
```

```
const cinterval rnd(cl_N x) {
  cinterval ans( rnd(x,down), rnd(x,up) );
  return ans;
}

const real rnd(cl_R x, const cl_R (*func)(cl_R)) {
  return stringConv(chop(x,func));
}

const cxsc::complex rnd(cl_N x, const cl_R (*func)(cl_R)) {
  cxsc::complex ans( rnd(realpart(x),func) , rnd(imagpart(x),func) );
  return ans;
}

const cl_R up(cl_R x) {
  return fceiling(x+One);
}

const cl_R down(cl_R x) {
  return ffloor(x-One);
}

const cl_R chop(cl_R x, const cl_R (*func)(cl_R)) {
  cl_R temp=x;
  real ans;
  int places2left=0, places;

  if ( x == Zero ) temp=Zero;
  else {
    int firstDigit=placeValue(x)-1;
    if ( firstDigit <= 0 ) {
      while ( temp > One ) {
        places2left++;
        temp=temp/Ten;
      }
      if ( places2left > SignificantDigits )
        screenErr("Too many digits to left of decimal point to convert.");
      places=SignificantDigits-places2left;
    }
    else places=SignificantDigits+firstDigit;
    temp=x;
    for ( int j=0; j<places; j++ ) temp=temp*Ten;
    temp=func(temp);
    for ( int j=0; j<places; j++ ) temp=temp/Ten;
  }
  return temp;
}

const real stringConv(cl_R x_cln) {
  char xStringArray[100];
  string xString;
  ostrstream outString(xStringArray,100);
  real x_cxsc;

  outString << x_cln << ends;
  xString=xStringArray;
  int position = xString.find('L');
  if ( position != string::npos ) xString[position]='E';
  xString >> x_cxsc;
  return x_cxsc;
}
```

B.5.2 CLN: calculation

Herewith a potpourri which makes up the main guts of cheby manipulation, including the routines to estimate the derivative of renormalization and to find the period 2 points.

```
const cl_N dotprod(cheby<cl_N,Deg>& prod1, cl_N prod2[]);
const cl_N dotprod(cl_N prod1[], cl_N prod2[]);
void svmult(cl_N vect[], cl_N foo, int i, int n);
void svmult(cl_N vect[], cl_R foo, int i, int n);
const cl_R vnorm(cl_N vect[], int i, int n);
const int placeValue(cl_R x);
// returns int value of first significant digit of x
const cl_R getEpsilon(cl_R x);
// returns 1.0e-(n+9) where n=placeValue(x)
const cl_N getLambda(int j, int n);
// compute angle for param
```

```
#ifdef _COMPLEX_
const cl_N sq(cheby<cl_N,Deg>& f, cl_N x);
// return f(f(x))
const cl_N D2(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df, cl_N x);
// return df(f(x))*df(x)
const cl_N getPer2Pt(cheby<cl_N,Deg>& f);
// return per2pt of f using newton's method
const cl_N f2MId(cheby<cl_N,Deg>& f, cl_N x);
// return f^2(x)-x
const cl_N getZero(cheby<cl_N,Deg>& f);
// return zero of f using newton's method
const cl_N justf(cheby<cl_N,Deg>& f, cl_N x);
// return f(x)
const cl_N falsePos(cheby<cl_N,Deg>& f, cl_N x1, cl_N x2,
  const cl_N (*func)(cheby<cl_N,Deg>& , cl_N));
// hacked up false position for root finding
const cl_N newton(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df, cl_N x1, cl_N x2,
 void (*func)(cheby<cl_N,Deg>& , chebyOdd<cl_N,Deg>& , cl_N, cl_N *, cl_N *));
// newton's method to find zero of func(f,df) between x1 and x2
void param(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& fix, cl_N unstable[],
  cl_N lambda);
// parametrize f=fix+lamba*unstable
void getf_j(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& fix, cl_N unstable[],int j,
  int n);
// get renormalized image of jth guy on arc
void av(cheby<cl_N,Deg>& f,cheby<cl_N,Deg>& f1, cheby<cl_N,Deg>& f2);
// set f=1/2(f1+f2) with constant term returned = 1;
#endif
#ifdef _REAL_
void DRCol(int j, cheby<cl_R, Deg>& f, cl_R& dRcol);
// computes jth col of partial derivates of DR(f)
void DR(cheby<cl_R,Deg>& f, cl_R dR[][Deg+1]);
// computes DR(f)
void print(cl_R dR[][Deg+1]);
void feig(cheby<cl_R,Deg>& f);
// sets f=feig
void psuedoNewton(cheby<cl_R,Deg>& f);
// does the Lanford psuedo-Newton method on f until Rf-f<Tolerance
#endif

const cl_N dotprod(cheby<cl_N,Deg>& prod1, cl_N prod2[]) {
  cl_N ans=cZero;
  for ( int i=0; i<=Deg; i++ ) ans=ans+prod1[i]*prod2[i];
  return ans;
}

const cl_N dotprod(cl_N prod1[], cl_N prod2[]) {
  cl_N ans=cZero;
  for ( int i=0; i<=Deg; i++ ) ans=ans+prod1[i]*prod2[i];
  return ans;
}

void svmult(cl_N vect[], cl_N foo, int i, int n) {
  // multiply the vector from pos i to pos n by scalar foo
  for ( int j=i; j<=n; j++ ) vect[j]=vect[j]*foo;
}

void svmult(cl_N vect[], cl_R foo, int i, int n) {
  // multiply the vector from pos i to pos n by scalar foo
  for ( int j=i; j<=n; j++ ) vect[j]=vect[j]*foo;
}

const cl_R vnorm(cl_N vect[], int i, int n) {
  // return the norm of the n-vector starting at the ith place
  cl_R ans=Zero;
  for ( int j=i; j<=n; j++ ) ans=ans+abs(vect[j])*abs(vect[j]);
  return sqrt(ans);
}

const int placeValue(cl_R x) {
  // returns smallest n s.t (10^n)x>1.
  int n=0;
  while ( abs(x) <= One ) {
    x=x*CAST(10.0);
    n++;
  }
  return n;
}

const cl_R getEpsilon(cl_R x) {
  // returns 10^-(n+offset)

  int n;
  cl_R epsilon=One;
  n=placeValue(x);
  for (int i=1; i<=n+9; i++) {
    epsilon=epsilon/CAST(10.0);
  }
  return epsilon;
}

const cl_N getLambda(int j, int n) {
  cl_R Pi=pi(precision);
  cl_N theta=cln::complex(Pi*Half+(CAST(j)*Half*Pi)/CAST(n),Zero);
  return Radius*exp(I*theta);
}

#ifdef _COMPLEX_

const cl_N sq(cheby<cl_N,Deg>& f, cl_N x) {
  cl_N fx=f(x);
  return f(fx);
}

const cl_N D2(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df, cl_N x) {
  cl_N fx=f(x);
  cl_N dfx=df(x);
  cl_N dffx=df(fx);
  return dffx*dfx;
}

const cl_N getPer2Pt(cheby<cl_N,Deg>& f) {
  // return the inside period 2 point using newton's method
  cl_N x1=-cHalf;
  cl_N x2=cZero;
  return falsePos(f,x1,x2,f2MId);
}

const cl_N f2MId(cheby<cl_N,Deg>& f, cl_N x) {
  // compute f^2(x)-x
  return sq(f,x)-x;
}

const cl_N getZero(cheby<cl_N,Deg>& f) {
  // find zeroes of f using newton's method
  cl_N x1=cHalf;
  cl_N x2=cHalf+cOne;
  return falsePos(f,x1,x2,justf);
}
```

```
const cl_N justf(cheby<cl_N,Deg>& f, cl_N x) {
  return f(x);
}

const cl_N falsePos(cheby<cl_N,Deg>& f, cl_N x1, cl_N x2,
  const cl_N (*func)(cheby<cl_N,Deg>& , cl_N)) {
  // find inside period two point of f using hacked up false position method

  cl_N fl,fh,xl,xh,swap,dx,del,funcVal,ans;

  fl=func(f,x1);
  fh=func(f,x2);

  if ( realpart(fl)*realpart(fh) > Zero )
    screenErr("Root not bracketed!");

  if ( realpart(fl) < Zero ) {
    xl=x1;
    xh=x2;
  } else {
    xl=x2;
    xh=x1;
    swap=fl;
    fl=fh;
    fh=swap;
  }
  dx=xh-xl;
  for ( int j=1; j<=200; j++) {
    ans=xl+dx*fl/(fl-fh);
    funcVal=func(f,ans);
    if ( realpart(funcVal) < Zero ) {
      del=xl-ans;
      xl=ans;
      fl=funcVal;
    } else {
      del=xh-ans;
      xh=ans;
      fh=funcVal;
    }
    dx=xh-xl;
    if ( abs(del) < Tolerance || abs(funcVal) < Tolerance ) {
      return ans;
    }
  }
  screenErr("Maximum number of iterations exceeded in root finding!");
}

const cl_N newton(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df, cl_N x1, cl_N x2,
 void (*func)(cheby<cl_N,Deg>& , chebyOdd<cl_N,Deg>& , cl_N, cl_N *, cl_N *)) {
  cl_N ans=Half*(x1+x2);
  for ( int j=0; j<1000; j++ ) {
    cl_N fx, dfx;
    func(f,df,ans,&fx,&dfx);
    cl_N dx=fx/dfx;
    ans=ans-dx;
    if ( (realpart(x1)-realpart(ans))*(realpart(ans)-realpart(x2)) < Zero )
      screenErr("Error. Jumped out of bracket in Newton.");
    if ( abs(dx) < Tolerance ) return ans;
  }
  screenErr("Error. Maximum number of iterations exceeded in newton.");
}

void param(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& fix, cl_N unstable[],
           cl_N lambda) {
  for ( int i=1; i<Deg+1; i++ )
    f[i]=fix[i]+lambda*unstable[i];
  setZeroCoeff(f);
}

void getf_j(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& fix, cl_N unstable[],int j,
            int n) {
  cl_N lambda=getLambda(j,n);
  param(f,fix,unstable,lambda);
  //  R(f,Renorms);
}

void av(cheby<cl_N,Deg>& f,cheby<cl_N,Deg>& f1, cheby<cl_N,Deg>& f2) {
  f=(f1+f2)*cHalf;
  setZeroCoeff(f);
}

void writePoem() {
// just to break up the monotony
  cout <<  "tomorrow hides i know not what tomorrow";
  cout << "will invite to fight the promise of i might. tomorrow";
```

```
   cout << "has a wicked waging war of why to ply";
   cout << "a wrinkle in, a sprinkle on, a sighing shame on you";
   cout << "on me! so strange and strangle sad now.";
   cout << "tomorrow waits to shed";
   cout << "to cuss and fuss and sow its shards";
   cout << "of sorrow like cut up broken beans. tomorrow's";
   cout << "shirt is black and laced with lack. tomorrow";
   cout << "spends in stones and stands unbraced";
   cout << "because the breeze is broken. tomorrow";
   cout << "frays and flays and frightens. tomorrow it will snow.";
   cout << "tomorrow has a will to whittle";
   cout << "and cultivates a little boney prize of lonely";
   cout << "because i did not know the no no of if only";
   cout << "could spread its curvy scurvy script across the graffiti";
   cout << "covered graveyard grimy of why me and";
   cout << "why oh silly boy did you ever look behind?";
}

#endif

#ifdef _REAL_

void DRCol(int j, cheby<cl_R,Deg>& f, cl_R dRcol[]) {
  // computes the row vector of dR/dfj and places it in dRCol.
  cheby<cl_R,Deg> rfPe(f),rfMe(f);
  cl_R epsilon=getEpsilon(f[j]);
  rfPe[j]=rfPe[j]+epsilon;              // computing partial wrt fj
  rfMe[j]=rfMe[j]-epsilon;
  setZeroCoeff(rfPe);
  setZeroCoeff(rfMe);
  R(rfPe);
  R(rfMe);
  for (int i=1; i<Deg+1; i++)           // use R(f+e_j)-R(f-e_j)/2e_j
    dRcol[i]=(rfPe[i]-rfMe[i])/(Two*epsilon);
}

void DR(cheby<cl_R,Deg>& f, cl_R dR[][Deg+1]) {
  for (int j=1; j<Deg+1; j++)
     DRCol(j,f,dR[j]);                 // rows and cols backward!
}

void print(cl_R dR[][Deg+1]) {
  cout << "s:={";
  for ( int i=1; i<Deg+1; i++ )
    for  ( int j=1; j<Deg+1; j++ ) {
      cout << "(" << i << "," << j << ")=" << dR[j][i]; // rows and cols bkwd
      if ( ( i == Deg ) && ( j == Deg ) )
        cout << "}:" << endl;
      else
        cout << ",";
    }
}

void feig(cheby<cl_R,Deg>& f) {
  f[1]=Half*Feig;
  setZeroCoeff(f);
}

void psuedoNewton(cheby<cl_R,Deg>& f) {
  cl_R J[Deg+1][Deg+1],test=One;
  cheby<cl_R,Deg> rf(f),temp;
  getJ(J);
  R(rf);
  temp=rf-f;
  while ( test>Tolerance ) {
    for ( int i=1; i<Deg+1; i++ ) {       // matrix multiply by J
      cl_R sum=Zero;
      for ( int j=1; j<Deg+1; j++ )
        sum=sum+J[j][i]*temp[j];
      temp[i]=sum;
    }
    f=f-temp;
    setZeroCoeff(f);
    rf=f;
    R(rf);
    temp=rf-f;
    test=Zero;
    for ( int i=1; i<Deg+1; i++) test=test+abs(temp[i]);
  }
}

#endif
```

B.5.3 CXSC: calculation

And next the interval equivalent.

```
interval Blo(interval orig, real eps);
cinterval Blo(cinterval orig, real eps);
void Blo(cicheby& orig, cicheby& blowguy, const real& eps);
int in(cicheby& small, cicheby& big);
cinterval dotprod(cicheby& prod1, civector prod2);
void svmult(civector& vect, cinterval foo, int i, int n);
const cinterval sq(cicheby& f, cinterval x);
// return f(f(x))
const cinterval sq(ccheby& f, cinterval x);
// return f(f(x))
const cinterval D2(cicheby& f, cichebyOdd& df, cinterval x);
// return df(f(x))*df(x)
void midishPt( ccheby& f, cicheby& F);
// return a ccheby which is the midpt of F to roundoff error
const cxsc::complex midishPt(cinterval x);
// return the midpt of x to round off error
const real midishPt(interval x);
// return the midpt of x to round off error
void hull(cicheby& ans, cicheby& f, cicheby& g);
// set ans[i]=hull(f[i],g[i])
const interval hull(interval x, interval y);
// return union (or hull) of two complex intervals
const cinterval hull(cinterval x, cinterval y);
// return union (or hull) of two real intervals
int in(cxsc::complex x, cinterval& X);
// check if x is in the interval X
const cinterval cinvl(cxsc::complex x, cxsc::complex y);
// return int [x,y] correctly!!! Order now irrelevant.
const interval invl(real x, real y);
// return int [x,y] correctly!!! Order now irrelevant.

interval Blo(interval orig, real eps) {
  return invl(Inf(orig)-abs(eps),Sup(orig)+abs(eps));
}

cinterval Blo(cinterval orig, real eps) {
  cinterval tmp ( Blo(Re(orig),eps) , Blo(Im(orig),eps) );
  return tmp;
}
```

```
void Blo(cicheby& orig, cicheby& blowguy, const real& eps) {

  Resize(blowguy, Lb(orig), Ub(orig));
  for ( int j=max(Lb(orig),1); j<=Ub(orig); j++ )
    blowguy[j]=Blo(orig[j],eps);
  setZeroCoeff(blowguy);
}

int in(cicheby& small, cicheby& big) {

  int ans=1;
  if ( (Lb(big)!=Lb(small)) || (Ub(big)!=Ub(small)) )
    screenErr("can't do in on different size chebys!");
  for (int j=max(Lb(big),1); j<=Ub(big); j++ )
    ans=ans*in(small[j],big[j]);
  return ans;
}


cinterval dotprod(cicheby& prod1, civector prod2) {
  // compute the vector dot product

  cinterval ans=_cinterval(0.0);
  for ( int i=0; i<=Deg; i++ ) ans=ans+prod1[i]*prod2[i];
  return ans;
}

void svmult(civector& vect, cinterval foo, int i, int n) {
  // multiply the vector from pos i to pos n by scalar foo

  for ( int j=i; j<=n; j++ ) vect[j]=vect[j]*foo;
}

const cinterval sq(cicheby& f, cinterval x) {
  cinterval fx=f(x);
  return f(fx);
}

const cinterval sq(ccheby& f, cinterval x) {
  cinterval fx=f(x);
  return f(fx);
}
```

```
const cinterval D2(cicheby& f, cichebyOdd& df, cinterval x) {
  cinterval fx=f(x);
  cinterval dffx=df(fx);
  cinterval dfx=df(x);
  return dffx*dfx;
}

void midishPt( ccheby& f, cicheby& F) {
  Resize(f,max(Lb(F),1),Ub(F));
  for ( int j=1; j<=Ub(f); j++ )
    f[j]=midishPt(F[j]);
  cinterval zeroTerm;
  zeroTerm=getZeroCoeff(f);
  if ( in(zeroTerm,F[0]) ) f.Tzero=zeroTerm;
  else
    screenErr("Zero interval not in hull!");
}

const cxsc::complex midishPt(cinterval x) {
  real ansR, ansI;
  ansR=midishPt(Re(x));
  ansI=midishPt(Im(x));
  complex ans( ansR, ansI );
  return ans;
}

const real midishPt(interval x) {
  real ans;
  ans=(Inf(x)+Sup(x))/2.0;
  if ( in(ans,x) ) return ans;
  else
    screenErr("Round error in computing midish point.");
}

void hull(cicheby& ans, cicheby& f, cicheby& g) {

  interval Zero=_interval(0.0);
  cinterval CZero(Zero,Zero);
  int start=max(Lb(f),Lb(g));
  int end=min(Ub(f),Ub(g));

  for ( int j=max(start,1); j<=end; j++ )
```

```
    ans[j]=hull(f[j],g[j]);
  if ( start > Lb(f) )
    for ( int j=Lb(f); j<start; j++ )
      ans[j]=hull(f[j],CZero);
  if ( start > Lb(g) )
    for ( int j=Lb(g); j<start; j++ )
      ans[j]=hull(g[j],CZero);
  if ( end < Ub(f) )
    for ( int j=end+1; j<=Ub(f); j++ )
      ans[j]=hull(f[j],CZero);
  if ( end < Ub(g) )
    for ( int j=end+1; j<=Ub(g); j++ )
      ans[j]=hull(g[j],CZero);
  setZeroCoeff(ans);
}

const cinterval hull(cinterval x, cinterval y) {
  cinterval ans;
  Re(ans)=hull(Re(x),Re(y));
  Im(ans)=hull(Im(x),Im(y));
  return ans;
}

const interval hull(interval x, interval y) {
  interval ans;
  Inf(ans)=min(Inf(x),Inf(y));
  Sup(ans)=max(Sup(x),Sup(y));
  return ans;
}

int in(cxsc::complex x, cinterval& X){
  return( in(Re(x),Re(X)) && in(Im(x),Im(X)) );
}

const cinterval cinvl(cxsc::complex x, cxsc::complex y) {
  return _cinterval(invl(Re(x),Re(y)),invl(Im(x),Im(y)));
}

const interval invl(real x, real y) {
  return _interval(min(x,y),max(x,y));
}

// the rest needs work ...
```

```
const cinterval getPer2Pt(ccheby& f, cichebyOdd& df) {
  // return the inside period 2 point using newton's method
  interval x1(-0.3,-0.2);
  interval x2( 0.0,0.1);
  cinterval x(  x1 , x2  );
  return newton(f,df,x,f2MId);
}

const cinterval f2MId(ccheby& f, cinterval x) {
  // compute f^2(x)-x
  return sq(f,x)-x;
}

void f2MId(ccheby& f, cichebyOdd& df, cinterval x, cinterval *fx,
           cinterval *dfx) {
  cxsc::complex One( 1.0 , 0.0 );
  cinterval efx,temp;

  efx=f(x);

  temp=f(efx);
  temp=temp-x;
  *fx=temp;

  temp=df(efx);
  temp=temp*df(x);
  temp=temp-One;
  *dfx=temp;
}

const cinterval getZero(ccheby& f) {
  // find zeroes of f using newton's method
  interval x1(0.5,1.0);
  interval x2(0.0,0.1);
  cinterval x(  x1, x2  );
  return falsePos(f,x,justf);
}

const cinterval getZero(ccheby& f, cichebyOdd& df) {
  // find zeroes of f using newton's method
  interval x1(0.5,1.0);
  interval x2(0.0,0.1);
  cinterval x(  x1, x2  );
  return newton(f,df,x,justf);
}

const cinterval justf(ccheby& f, cinterval x) {
  return f(x);
}

void justf(ccheby& f, cichebyOdd& df, cinterval x, cinterval *fx,
           cinterval *dfx) {
  *fx=f(x);
  *dfx=df(x);
}

const cinterval newton(ccheby& f, cichebyOdd& df, cinterval x,
  void (*func)(ccheby& , cichebyOdd& , cinterval, cinterval *, cinterval *)) {

  for ( int j=0; j<1000; j++ ) {
    cinterval fx, dfx;
    func(f,df,x,&fx,&dfx);
    x=x-fx/dfx;
    cout << x << endl;
    if ( abs(Sup(x)-Inf(x)) < 1.0e-10 ) return x;
  }
  screenErr("Error. Maximum number of iterations exceeded in newton.");
}

const cinterval falsePos(ccheby& f, cinterval x,
  const cinterval (*func)(ccheby& , cinterval)) {
  // find inside period two point of f using hacked up false position method

  real dx;
  cxsc::complex testPt, vertex, Zero( 0.0 , 0.0 );
  cinterval fx, testInt, ftestInt;
  fx=func(f,x);
  real Tolerance=1.0e-15;

  if ( !(in(Zero,fx)) )
    screenErr("Root not bracketed!");
  dx=abs(Sup(x)-Inf(x));
  for ( int j=1; j<=200; j++) {
    //testPt=Inf(x)+dx*(Inf(fx)/(Inf(fx)-Sup(fx))); // false pos.
    //if ( !(in(testPt,x)) ) testPt=midishPt(x);    // if not, bisectio
```

```
      testPt=midishPt(x);
      cout << SetPrecision(25,25) << x << endl;
      cout << SetPrecision(25,25) << testPt << endl;
      if ( !(in(testPt,x)) )
        screenErr("testPt not in interval!");
      vertex=_complex( Inf(Re(x)) , Inf(Im(x)) );
      testInt=cinvl(testPt,vertex);
      ftestInt=func(f,testInt);
      if ( in(Zero,ftestInt) ) x=testInt;
      else {
        vertex=_complex( Inf(Re(x)) , Sup(Im(x)) );
        testInt=cinvl(testPt,vertex);
        ftestInt=func(f,testInt);
        if ( in(Zero,ftestInt) ) x=testInt;
        else {
          vertex=_complex( Sup(Re(x)) , Inf(Im(x)) );
          testInt=cinvl(testPt,vertex);
          ftestInt=func(f,testInt);
          if ( in(Zero,ftestInt) ) x=testInt;
          else {
            vertex=_complex( Sup(Re(x)) , Sup(Im(x)) );
            testInt=cinvl(testPt,vertex);
            ftestInt=func(f,testInt);
            if ( in(Zero,ftestInt) ) x=testInt;
            else
              screenErr("Lost zero in falsePos!");
          }
        }
      }
      dx=abs(Sup(x)-Inf(x));
      fx=func(f,x);
      if ( dx < Tolerance  )
        return x;
    }
  screenErr("Maximum number of iterations exceeded in root finding!");
}
```

B.5.4 Graphing

We give the code for drawing M-sets.

```
struct coordStruct {
  int x;                                  //  0=not from spiralling
  int y;                                  // -1=not in big grid
};

struct spiralStruct {
  coordStruct coord;
  int armLength;
  int horizontal;                         // +1/-1 for horiz/vert
  int forward;                            // +1/-1 for pos/neg
  int i;                                  // running variable
  bool done;
};

void drawMandelbrot(char* file, pSeries& w, int boxesPerRow,
  int maxIterates, int renorms);
spiralStruct getSpiralStartState(int BoxesPerRow);
int doBox(coordStruct coord, pSeries& w, int BoxesPerRow,
  int MaxIterates, int renorms);
void printSpiral(spiralStruct spiral);

void drawMandelbrot(char* file, pSeries& w, int boxesPerRow,
  int maxIterates, int renorms) {

  if ( boxesPerRow/2 == 0 ) {
    cout << "Boxes per row must be even, setting it to be 501!" << endl;
    boxesPerRow = 501;
  }

  int grid[boxesPerRow][boxesPerRow];
  // initialize grid
  for ( int i=0; i<boxesPerRow; i++ ) {
    for (int j=0; j<boxesPerRow; j++ ) {
      grid[i][j]=0;
    }
  }

  unsigned int end=boxesPerRow*boxesPerRow;
  unsigned int printNow=end/1000;
```

```
    unsigned int printCount=0;

    // the basic plot: spiral through small grid, checking each box

    unsigned int boxesDone=0;
    spiralStruct spiral=getSpiralStartState(boxesPerRow);

    while ( !spiral.done ) {
      while ( spiral.i < spiral.armLength ) {
        if ( printCount == printNow ) {
  #include "mandelbrot.inl"
          printCount=0;
        }
        grid[spiral.coord.x-1][spiral.coord.y-1]=
          doBox(spiral.coord,w,boxesPerRow,maxIterates,renorms);
        boxesDone++;
        printCount++;
        if ( spiral.horizontal==1 ) {      // move horizontally
          spiral.coord.x=spiral.coord.x+spiral.forward;
        } else {                           // move vertically
          spiral.coord.y=spiral.coord.y+spiral.forward;
        }
        if ( boxesDone == end ) {
          spiral.done=true;                // check to see if done
          spiral.i=spiral.armLength;
        }
        spiral.i++;
      }
      spiral.horizontal*=-1;               // switch direction
      if ( spiral.horizontal==1 ) {        // if completed vertical move then
        spiral.forward*=-1;                // switch sign
        spiral.armLength++;                // and increment spiral arm length
      }
      spiral.i=0;                          // NB: reset length traveled on arm
    }

#include "mandelbrot.inl"

}

void initializeGrid(int boxesPerRow, int& grid) {
  // set all entries to 0
```

```
}

spiralStruct getSpiralStartState(int boxesPerRow) {

  spiralStruct spiral;

  spiral.armLength=spiral.horizontal=spiral.forward=1;
  spiral.i=0;
  spiral.coord.x=spiral.coord.y=(boxesPerRow+1)/2;// start in center square
  spiral.done=false;
  return spiral;
}

int doBox(coordStruct coord, pSeries& w, int boxesPerRow, int maxIterates,
  int renorms) {
  // what to do when you arrive in a new box in the grid

  cheby<cl_N,Deg> f;

  // compute function lying at center of box

  // offset so center is (0,0)
  int center,i,j;
  i=coord.x;
  j=coord.y;
  center=(boxesPerRow+1)/2;
  i-=center;
  j-=center;
  // compute grid size and make call to parametrization
  //cl_R halfBoxSize=(One/CAST((boxesPerRow-1)/2))/Two;
  cl_R halfBoxSize=(One/CAST((boxesPerRow-1)/2));
  cl_N t;
  t=cln::complex(CAST(i)*halfBoxSize,CAST(j)*halfBoxSize);
  T(w,f,t);

  //follow the critical orbit

  cl_N x=cZero;

  if ( renorms != 4 ) {

    // only doing polys-like for renorms=4, so just follow critical orbit
```

```
      for ( int i=0; i<maxIterates; i++ ) {
        x=f(x);
      if ( abs(x) > Two ) return i;
      }
      return maxIterates+1;
    }

    else {

    // must look for poly-type guys only

      cl_R radius;
      radius=abs(getPer2Pt(f));
      for ( int i=1; renorms*i<=maxIterates; i++ ) {
        for ( int j=1; j<renorms; j++ ) {
          x=f(x);
          if ( abs(x) < radius ) {
            return renorms*(i-1)+j;
          }
        }
        x=f(x);
        if ( abs(x) > radius ){
          return renorms*i;
        }
      }
      return maxIterates+1;
    }
  }

  void printSpiral(spiralStruct spiral) {
    // print out the state of a spiral

    cout  << "SPIRAL STATE\n";
    cout << "      coord=(" << spiral.coord.x << "," << spiral.coord.y << ")\n";
    cout << "  armLength= " << spiral.armLength << "\n";
    cout << " horizontal= " << spiral.horizontal << "\n";
    cout << "    forward= " << spiral.forward << "\n";
    cout << "          i= " << spiral.i << "\n";
    cout << "       done= " << spiral.done<< "\n\n";
  }

// essentially the write file subroutine
// Hack way of getting something done! Don't know how to pass double array.

// write File
ofstream fout(file);
if ( !fout.good()) screenErr("Can't open file");

// write pbm header

fout << "P2\n";
fout << boxesPerRow << " " << boxesPerRow << " # width by height" << endl;
fout << maxIterates+2 << " # maximum value allowed in array" << endl << endl;

// write spiral state as a comment

fout << "# SPIRAL STATE:" << endl;
fout << "#      coord=(" << spiral.coord.x << "," << spiral.coord.y << ")"
     << endl;
fout << "#  armLength= " << spiral.armLength << endl;
fout << "# horizontal= " << spiral.horizontal << endl;
fout << "#    forward= " << spiral.forward << endl;
fout << "#          i= " << spiral.i << endl;
fout << "#       toDo= " << end-boxesDone << endl;
fout << "#       done= " << spiral.done<< endl << endl;

fout << "# MAX ESCAPE" << endl << endl;

// write data

for ( int j=boxesPerRow-1; j>=0; j-- ) {
  for (int i=0; i<boxesPerRow; i++ ) {
    fout << grid[i][j] << " ";
  }
  fout << endl;
   }

fout << endl;
fout.close();
```

B.5.5 The Estimating Lemma

Finally, the code to carry out the program of the Estimating Lemma.

```
void iproof(int start, int end, int incr, int N, char *file);
const real getDerivBound(cicheby& f, cichebyOdd& df, cinterval per2f);
// compute min of |(f^2)'| on a disk of radius per2radius about per2f;
// increase accuracy to decrease the bound
const real getSquareBound(cicheby& f, cicheby& g, cinterval per2f);
// compute max of |f^2-g^2| on a disk of radius per2radius about per2f
const real estDerivBound(cicheby& f, cichebyOdd& df, cinterval per2f);
// compute deriv bound at 100 points
const real estSquareBound(cicheby& f, cicheby& g, cinterval per2f);
// compute square bound at 100 points.

void proof(int start, int end, int incr, int N);
const cl_R getDerivBound(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df,
  cl_N per2f);
// compute min of |(f^2)'| on a disk of radius per2radius about per2f;
// increase accuracy to decrease the bound
const cl_R estDerivBound(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df,
  cl_N per2f);
// compute deriv bound at 100 points
const cl_R estSquareBound(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& g, cl_N per2f);
// compute square bound at 100 points

void iproof(int start, int end, int incr, int N, char *file) {

  cheby<cl_N,Deg> f1,f2,fix;
  cl_N unstable[26];
  cxsc::complex cOne( 1.0, 0.0);
  cinterval ciOne=_cinterval(cOne);

  getFix(fix);
  getUnstable(unstable);

  if ( incr == 1 ) getf_j(f2,fix,unstable,start,N);

  for (int j=start; j<end; j=j+incr) {

    if ( incr > 1) {
      getf_j(f1,fix,unstable,j,N);
      getf_j(f2,fix,unstable,j+1,N);
} else if ( incr == 1 ) {
  f1=f2;
  getf_j(f2,fix,unstable,j+1,N);
} else
  screenErr("incr must be >= 1!");

cheby<cl_N,Deg> f;
cicheby cif1,cif2,cif,cig;
cichebyOdd cidf;
av(f,f1,f2);
rnd(cif,f);
rnd(cif1,f1);
rnd(cif2,f2);
hull(cig,cif1,cif2);
D(cidf,cif);

int degf=deg(f);
int degf2=degf*degf;

cl_N per2=getPer2Pt(f);
cinterval ciper2=rnd(per2);
cl_N zero=getZero(f);
cinterval cizero=rnd(zero);

cinterval af=cif(ciOne);
cinterval bf=cif(af);
cinterval cf=cif(bf);
cinterval ag=cig(ciOne);
cinterval bg=cig(ag);
cinterval fag=cif(ag);
cinterval fzero=cif(cizero);
cinterval dfzero=cidf(cizero);

real df2af=AbsMax(abs(D2(cif,cidf,af)));
real df2p2M1=AbsMin(abs(D2(cif,cidf,ciper2)-ciOne));
real zeroAdj=mulu((degf),AbsMax(abs(fzero/dfzero)));
cinterval adjzero=Blow(cizero,zeroAdj);

real M1=AbsMax(abs(af-ag));  // must bump
real M2=AbsMax(abs(cif(fag)-cig(bg)));
real M3=getSquareBound(cif,cig,ciper2); // must bump
real eps=getDerivBound(cif,cidf,ciper2);
real delta=AbsMax(abs(sq(cif,ciper2)-ciper2));  // must bump
```

```
  real A=divu(mulu(delta,degf2),df2p2M1);
  real B=addu(mulu(mulu(4.0,M1),df2af),M2);
  real C=divu(mulu(8.0,M3),eps);
  real E=subd(AbsMin(abs(cf)),AbsMax(abs(ciper2)));
  real surplus=subd(E,addu(A,addu(B,C)));

  // checks

  if ( !( eps>0.0 ))
    fileErr(file,"epsilon not positive!");
  if ( !( delta>0.0 ))
    fileErr(file,"delta not positive!");
  if ( !( A>0.0 ))
    fileErr(file,"A not positive!");
  if ( !( B>0.0 ))
    fileErr(file,"B not positive!");
  if ( !( C>0.0 ))
    fileErr(file,"C not positive!");
  if ( !( E>0.0 ))
    fileErr(file,"E not positive!");
  if ( !( M1>0.0 ))
    fileErr(file,"M1 not positive!");
  if ( !( M2>0.0 ))
    fileErr(file,"M2 not positive!");
  if ( !( M3>0.0 ))
    fileErr(file,"M3 not positive!");
  if ( !( surplus>0.0 ))
    fileErr(file,"surplus not positive!");

  if ( !( addu(A,C)<AbsMin(iPer2radius) ) )
    fileErr(file,"A+C bigger than per2 radius!");
  if ( !( AbsMin(abs(af))>M1 ) )
    fileErr(file,"0 in D(a_f,M1)!");
  if ( !( AbsMin(abs(abs(adjzero)-abs(af)))>M1 ) )
    fileErr(file,"f^{-1}(0) (pos or neg) in D(a_f,M1)!");
  if ( !( AbsMin(abs(abs(adjzero)-abs(ciper2)))>AbsMax(iPer2radius) ) )
    fileErr(file,"f^{-1}(0) (pos or neg) in D(per2,per2radius)!");

  if ( incr>1 ) {

    ofstream fout(file, ios::app);
    if ( !fout.good() ) {
        string fileErr="Can't open ";
        fileErr.append(file);
        screenErr(fileErr);
      }
      fout << "For j=" << j << endl;
      fout << SetPrecision(50,50);
      fout << "A =" << A << endl;
      fout << "B =" << B << endl;
      fout << "C =" << C << endl;
      fout << "E =" << E << endl;
      fout << "M1=" << M1 << endl;
      fout << "M2=" << M2 << endl;
      fout << "M3=" << M3 << endl;
      fout << "e =" << eps << endl;
      fout << "d =" << delta << endl;
      fout << "E-(A+B+C)=" << surplus << endl;
      fout << "******************************************************" << endl;
      fout.close();

    } else {
      ofstream fout(file, ios::app);
      if ( !fout.good() ) {
        string fileErr="Can't open ";
        fileErr.append(file);
        screenErr(fileErr);
      }
      fout << j << " ";
      fout.close();
    }

    if ( j == end-incr && incr>1 ) j--;

  }
}

const real getDerivBound(cicheby& f, cichebyOdd& df, cinterval per2f) {

  // g(z)=iPer2radius*z+per2f  --   map unit circle to D(per2f,per2fradius)
  // p(z)=(f^2)'(g(z))
  // trying to bound |p| from below on unit circle
  // n=deg of (p)'
  // m >> 2*n
  // L=max{|p(z)|: z a 2nth root of unity}
```

```
    // M=min{|p(z)|: z an m'th root of unity}
    // thm: max mod p <= n*L;
    // cor: min{|p(z)|: z on unit circle}>=M-Pi*(n/m)*L

    int n=deg(f)*deg(f)-1;      // deg of (f^2)'
    int m=DerivBoundAcc*2*n;    // m >> 2*n
    interval Pi( acos(_interval(-1.0)) );
    cxsc::complex I(0,1);
    real L=0.0;
    real M=0.0;

    // compute max at 2nth roots of unity
    for ( int k=1; k<=2*n; k++ ) {
      interval t=Pi*(k/_interval(n));          // a 2nth root of unity
      cinterval z=iPer2radius*exp(I*t)+per2f; // g(that root)
      interval  pz=abs(D2(f,df,z))    ;        // |p(z)|
      if ( AbsMax(pz) > L ) L=AbsMax(pz);
    }

    // compute min at m'th roots of unity
    for ( int k=1; k<=m; k++ ) {
      interval t=Pi*(2*k/_interval(m));        // an m'th root of unity
      cinterval z=iPer2radius*exp(I*t)+per2f; // g(that root)
      interval  pz=abs(D2(f,df,z))    ;        // |p(z)|
      if ( AbsMin(pz) < M || k == 1 ) M=AbsMin(pz);
    }

    // corollary: M-Pi*(n/m)*L
    real ans=subd(M,mulu(AbsMax(Pi),mulu(L,divu(n,m))));
    return subd(ans,1.0);
}

const real getSquareBound(cicheby& f, cicheby& g, cinterval per2f) {
 // trying to bound |f^2(z)-g^2(z)| from above
  // c(z)=iPer2radius*z+per2f  --   map unit circle to D(per2f,per2fradius)
  // p(z)=f^2(c(z))-g^2(c(z))
  // q(t)=|p(e^(it))|^2        --   turn into a *real* problem!

  // N = deg q = deg p = (deg f)^2 = (deg g)^2
  int N=deg(f)*deg(f);

  // divide [0,2*Pi] into I_1,...,I_n
  // |I_k|=2*h, need h<Pi/N
```

```
    // need n>2*N
    //  h=|I_k|/2=(2*Pi/n)/2=Pi/n
    int n=SquareBoundAcc*2*N;
    interval Pi( acos(_interval(-1.0)) );
    interval h=Pi/_interval(n);
    if ( !( AbsMax(h)<AbsMin(Pi/_interval(N)) ) )
      screenErr("h not small enough for steckin!");

    cxsc::complex I(0,1);
    real max=0.0;

    // t_k=midpoint of I_k
    // compute q~=max{q(t_k): k=1,...,n}
    for ( int k=1; k<=n; k++ ) {
      interval  t=_interval(2*k-1)*h;          // t=midpoint of I_k=h+2(k-1)h
      cinterval z=iPer2radius*exp(I*t)+per2f; // z=c( e^(I*t) )
      interval  q=abs(sq(f,z)-sq(g,z));
      q=q*q;                                   // q=|f^2-g^2|^2
      if ( AbsMax(q) > max ) max=AbsMax(q);
    }

    // Steckin's lemma: ||q||<=q~*sec(N*h)
    return AbsMax(sqrt(max/cos(N*h)));         // sqrt because we want p not q
  }

  const real estDerivBound(cicheby& f, cichebyOdd& df, cinterval per2f) {
    interval Pi( acos(_interval(-1.0)) );
    cxsc::complex I(0,1);
    real min=0.0;
    // compute min at 100 points
    for ( int k=0; k<100; k++ ) {
      interval  t=2.0*Pi*k/_interval(100);
      cinterval z=iPer2radius*exp(I*t)+per2f;
      interval ans=abs(D2(f,df,z));
      if ( AbsMin(ans) < min || k == 1 ) min=AbsMin(ans);
    }
    return min-1.0;
  }

  const real estSquareBound(cicheby& f, cicheby& g, cinterval per2f) {
    interval Pi( acos(_interval(-1.0)) );
    cxsc::complex I(0,1);
    real max=0.0;
  // compute max at 100 points
  for ( int k=0; k<100; k++ ) {
    interval  t=2.0*Pi*k/_interval(100);
    cinterval z=iPer2radius*exp(I*t)+per2f;
    interval  ans=abs(sq(f,z)-sq(g,z));
    if ( AbsMax(ans) > max ) max=AbsMax(ans);
  }
  return max;
}

void proof(int start, int end, int incr, int N) {
  cheby<cl_N,Deg> f,g,fix;
  chebyOdd<cl_N,Deg> df;
  cl_N unstable[26];
  cl_N per2,zero;

  getFix(fix);
  getUnstable(unstable);

  if ( incr == 1 ) getf_j(g,fix,unstable,start,N);

  for (int j=start; j<=end; j=j+incr) {

    if ( j == end && incr>1 ) j--;

    if ( incr > 1) {
      getf_j(f,fix,unstable,j,N);
      getf_j(g,fix,unstable,j+1,N);
    } else if ( incr == 1 ) {
      f=g;
      getf_j(g,fix,unstable,j+1,N);
    } else
      screenErr("incr must be >= 1!");

    D(df,f);

    cl_N per2f=getPer2Pt(f);
    cl_N zero=getZero(f);

    cl_N af=f(cOne);
    cl_N bf=f(af);
    cl_N cf=f(bf);
```

```
      cl_N ag=g(cOne);
      cl_N bg=g(ag);

      cl_N fag=f(ag);

      cl_N dfbf=df(bf);
      cl_N dfaf=df(af);

      int degf=deg(f);
      int degf2=degf*degf;

      cl_R M1=abs(af-ag);   // ADD SMALL SOON!!!
      cl_R M2=abs(f(fag)-g(bg));

      cl_R M3=estSquareBound(f,g,per2f);
      cl_R eps=estDerivBound(f,df,per2f);

      cl_R A=Tolerance*CAST(degf2)/(abs(D2(f,df,per2)-cOne));
      cl_R B=Four*M1*abs(dfbf*dfaf)+M2;
      cl_R C=Two*Four*(M3/eps);  // ADD SMALL SOON !!!
      cl_R E=abs(cf)-abs(per2f);

      cout << "For j=" << j << endl << endl;
      cout << "A =" << A << endl;
      cout << "B =" << B << endl;
      cout << "C =" << C << endl;
      cout << "E =" << E << endl;
      cout << "M1=" << M1 << endl;
      cout << "M2=" << M2 << endl;
      cout << "M3=" << M3 << endl;
      cout << "e =" << eps << endl;
      cout << "E-(A+B+C)=" << E-(A+B+C) << endl;
      cout << "*******************************************************" << endl;

      if ( j == end-1 && incr>1 ) j=end+1;

  }
}

const cl_R getDerivBound(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df,
  cl_N per2f) {

  // g(z)=Per2radius*z+per2f  --   map unit circle to D(per2f,Per2fradius)
```

```
    // p(z)=(f^2)'(g(z))
    // trying to bound |p| from below on unit circle
    // n=deg of (p)'
    // m >> 2*n
    // L=max{|p(z)|: z a 2nth root of unity}
    // M=min{|p(z)|: z an m'th root of unity}
    // thm: max mod p <= n*L;
    // cor: min{|p(z)|: z on unit circle}>=M-Pi*(n/m)*L

    int n=deg(f)*deg(f)-1;          // deg of (f^2)'
    int m=DerivBoundAcc*2*n;        // m >> 2*n
    cl_R Pi=pi(precision);
    cl_N cPi=cln::complex(Pi,Zero);
    cl_R L=Zero;
    cl_R M=Zero;

    // compute max at 2nth roots of unity
    for ( int k=1; k<=2*n; k++ ) {
      cl_N t=I*cPi*CAST(k)/CAST(n);       // a 2nth root of unity
      cl_N z=Per2radius*exp(t)+per2f;   // g(that root)
      cl_R pz=abs(D2(f,df,z));        // |p(z)|
      if ( pz > L ) L=pz;
    }

    // compute min at m'th roots of unity
    for ( int k=1; k<=m; k++ ) {
      cl_N t=I*cPi*cTwo*CAST(k)/CAST(m);  // an m'th root of unity
      cl_N z=Per2radius*exp(t)+per2f;   // g(that root)
      cl_R pz=abs(D2(f,df,z));        // |p(z)|
      if ( pz < M || k == 1 ) M=pz;
    }

    return (M - Pi*(CAST(n)/CAST(m))*L) - One;  // corollary
}

const cl_R estDerivBound(cheby<cl_N,Deg>& f, chebyOdd<cl_N,Deg>& df,
  cl_N per2f) {
  cl_N cPi=cln::complex(pi(precision),Zero);
  cl_R min=Zero;
  // compute min at 100 pts
  for ( int k=0; k<100; k++ ) {
    cl_N t=I*cPi*CAST(2*k)/CAST(100);
    cl_N z=Per2radius*exp(t)+per2f;
    cl_R ans=abs(D2(f,df,z));
    if ( ans < min || k==1 ) min=ans;
  }
  return min-One;
}

const cl_R estSquareBound(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& g, cl_N per2f) {
  cl_N cPi=cln::complex(pi(precision),Zero);
  cl_R max=Zero;
  // compute max at 100 pts
  for ( int k=0; k<100; k++ ) {
    cl_N t=I*cPi*CAST(2*k)/CAST(100);
    cl_N z=Per2radius*exp(t)+per2f;
    cl_R ans=abs(sq(f,z)-sq(g,z));
    if ( ans > max ) max=ans;
  }
  return max;
}
```

B.6 Concluding Remarks

Despite the length of the above, we have omitted much, including all the various "main" (control) programs which call the above routines. These are completely straightforward and consist chiefly of function calls. As an example, we cite the code for finding the unstable manifold:

```
pSeries w,v;
cheby<cl_N,Deg> fix,temp;
cl_N unstable[26];
getFix(fix);
getUnstable(unstable);
bool keepgoing=true;
int iter=10;

#include "dat/w.dat"
```

```
    cl_R ans=abs(D2(f,df,z));
    if ( ans < min || k==1 ) min=ans;
  }
  return min-One;
}

const cl_R estSquareBound(cheby<cl_N,Deg>& f, cheby<cl_N,Deg>& g, cl_N per2f) {
  cl_N cPi=cln::complex(pi(precision),Zero);
  cl_R max=Zero;
  // compute max at 100 pts
  for ( int k=0; k<100; k++ ) {
    cl_N t=I*cPi*CAST(2*k)/CAST(100);
    cl_N z=Per2radius*exp(t)+per2f;
    cl_R ans=abs(sq(f,z)-sq(g,z));
    if ( ans > max ) max=ans;
  }
  return max;
}
```

B.6 Concluding Remarks

Despite the length of the above, we have omitted much, including all the various "main" (control) programs which call the above routines. These are completely straightforward and consist chiefly of function calls. As an example, we cite the code for finding the unstable manifold:

```
  pSeries w,v;
  cheby<cl_N,Deg> fix,temp;
  cl_N unstable[26];
  getFix(fix);
  getUnstable(unstable);
  bool keepgoing=true;
  int iter=10;

  #include "dat/w.dat"
```

```
    while ( keepgoing ) {
      iter++;
      v=w;
      cl_R change=Zero;
      if ( Renorms > 1 ) {
        Rlambda(w,Renorms);
        R(w,Renorms-1);
      }
      else R(w);
      change=compare(v,w);
      formatPrint(w,iter,change,"pseries.dat");
    }
  }
```

Other omitted code includes some error reporting routines, various interval versions of power series approximations to capture the unstable manifold, and the definitions of *fix*, *feig*, *unstable*, etc., the numerics of which are mostly reported in the tables at the end of Chapter 5.

Bibliography

[CE] M. Campanino, H. Epstein, *On the existence of Feigenbaum's fixed point*, Commun. Math. Phys. **79** (1981) 261-302.

[CER] M. Campanino, H. Epstein, D. Ruelle, *On Feigenbaum's functional equation*, Topology **21** (1982) 125-129.

[CoE] P. Collet, J.-P. Eckmann, *Iterated maps on the interval as dynamical systems*, Birkhäuser, Boston, 1980.

[CT1] P. Coullet, C. Tresser, *Itérations d'endomorphismes et groupe de renormalisation*, J. Phys. Colloque C 539, C5-25 (1978)

[CT2] P. Coullet, C. Tresser *Itérations d'endomorphismes et groupe de renormalisation*, C.R. Acad. Sci. Paris **287A** (1978), 577-580

[Cv] P. Cvitanovic, *Universality in chaos (or, Feigenbaum for cyclists)*, Preprint NORDITA, Copenhagen, Denmark 1983.

[DGP] B. Derrida, A. Gervois, Y. Pomeau, *Universal metric properties of bifurcations of endomorphisms*, J. Phys. A. **12** (1979), 269-296.

[D] A. Douady, *Julia sets and the Mandelbrot set*, The beauty of fractals: images of complex dynamical systems, edited by H.-O. Peitgen and P. H. Richter, Springer-Verlag, 1986, 161-173.

[DH] A. Douady, J. Hubbard, *On the dynamics of polynomial-like mappings*, Ann. Sci. Ec. Norm. Sup. (Paris) **18** (1985) 287-343.

[EE] J.P. Eckmann, H. Epstein, *Bounds on the unstable eigenvalue for period doubling*, Comm. Math. Phys. **128** (1990), 427-435.

[EW] J.-P. Eckmann, P. Wittwer, *A complete proof of the Feigenbaum conjectures*, J. Statist. Phys. **46** (1987), 455-475.

[E1] H. Epstein, *Fixed points of composition operators II*, Nonlinearity **2** (1989), 305-310.

[E2] H. Epstein, *Fixed points of the period-doubling operator* Lecture Notes, Lausanne, (1992).

[GS1] J. Graczyk, G. Świątek, *Generic hyperbolicity in the logistic family*, Ann. of Math. (2) **146** (1997), no. 1, 1-52.

[GS2] J. Graczyk, G. Świątek, *The real Fatou conjecture*, Ann. of Math. Studies **144** Princeton University Press, 19998

[F1] M.J. Feigenbaum, *Quantitative universality for a class of non-linear transformations*, J. Statist. Phys. **19** (1978) 25-52.

[F2] M.J. Feigenbaum, *The universal metric properties of non-linear transformations*, J. Statist. Phys. **21** (1979) 669-706.

[F3] M.J. Feigenbaum, *The transition to aperiodic behavior in turbulent sytems*, Comm. Math. Phys. **77** (1980) 65-86.

[F4] M.J. Feigenbaum, *Universal behavior in non-linear systems*, Los Alamos Sci. **1** (1980) 4-27.

[GSK] A.I. Golberg, Ya. G. Sinai, K.M. Khanin *Universal properties of a sequece of period-tripling bifurcations*, Russian Math. Surveys **38** (1983), 187-88.

[G] J.J. Green, *Calculating the maximum modulus of a polynomial using Stečkin's Lemma*, SIAM Journal of Numerical Analysis, **36**, Number 4, (1999) 1022-1029

[H] P. Henrici *Applied and computational complex analysis vol 1*, John Wiley & Sons, NY (1974) 467-8.

[Hu] J.H. Hubbard, *Local connectivity of the Julia sets and bifurcation loci: three theorems of J.-C. Yoccoz*, Topological methods in modern mathematics, A symposium in honor of John Milnor's 60th birthday, Publish or Perish, Houston, 1993.

[KSvS] O. Kozlovski, W. Shen, S. van Strien, *Density of hyperbolicity in dimension one*, Preprint, http://www.maths.warwick.ac.uk/~strien/Publications/axioma30july.pdf, July 2004

[L1] O. E. Lanford, *A computer assisted proof of the Feigenbaum conjectures*, Bull. Amer. Math. Soc. **6** (1982), 427-434.

[L2] O. E. Lanford, *A shorter proof of the existence of the Feigenbaum fixed point*, Commun. Math. Phys. **96** (1984) 521-538.

[L3] O.E. Lanford, *Computer Assisted Proofs*, Computational Methods in Field Theory: Proceedings of the 31. Internationale Universitätswochen für Kern- und Teilchenphysik, Schladming, Austria, February 1992, edited by H. Gausterer and C. B. Lang, Springer Lecture Notes in Physics **409** (1992), 43-58.

[Le] T. Lei, *Similarity between the Mandelbrot set and Julia Sets*, Communications in Mathematical Physics **134** (1990), 587-617.

[Ly1] M. Lyubich, *Feigenbaum-Coullet-Tresser universality and Milnor's Hairiness Conjecture*, Ann. of Math **149** (1999), 319-420.

[Ly2] M. Lyubich, *Renormalization ideas in conformal dynamics*, Current Developments in Mathematics, 1995, (eds: R Bott et al), International Press, Cambridge, MA, 1994, 155-190.

[Ly3] M. Lyubich, *Dynamics of quadratic polynomials, I, II*, Acta. Math **178** (1997), 185-297.

[Ly4] M. Lyubich, *Almost ever real quadratic map is either regular or stochastic*, preprint IMS at Stony Brook, #1997/8.

[McM1] C. McMullen, *Copmlex dynamics and renormalization*, Annals of Math. Studies **135**, (1994).

[McM2] C. McMullen, *Renormalization and 3-manifolds which fiber over the circle*, Annals of Math. Studies **142**, Princeton University Press, 1996.

[MS] W. de Melo, S. van Strien, *One-dimensional dynamics*, Springer-Verlag, New York, 1993.

[MSS] M. Metropolis, M.L. Stein, and P.R. Stein, *On finite limit sets for transformations of the unit interval*, J. Combin. Theory Ser. A **15** (1973), 25-44.

[M1] J. Milnor, *Local connectivity of Julia sets: Expository Lectures* Stony Brook IMS preprint 1992/11.

[M2] J. Milnor, *Periodic orbits, external rays and the Mandelbrot set: an expository account* Stony Brook IMS preprint 1999/3

[M3] J. Milnor, *Self-Similarity and Hairiness in the Mandelbrot Set*, Computers in Geometry and Topology, M. Tangora (editor), Lect. Notes Pure Appl. Math. 114, Dekker 1989, 211-257.

[MMR] G.V. Milovanovic, D.S. Mitrinovic, and Th. M. Rassias, *Topics in polynomials*, World Scientific (1994)

[PFTV] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, *Numerical recipes in C: the art of scientific computing*, Cambridge University Press, New York, 1990

[S1] D. Sullivan, *Quasiconformal homeomorphisms in dynamics, topology and geometry*, Proc. ICM-86, Berkeley **II**, A.M.S. (1987), 1216-1228.

[S2] D. Sullivan, *Bounds, quadratic differentials, and renormalization conjectures*, AMS Centennial Publications **II**, Mathematics into Twenty-first Century, 417-466, 1992.

[T] C. Tresser, *Aspects of renormalization in dynamical systems theory*, Chaos and comlexity, 11-19, Editions Frontiers, 1995.

[VSK] E. B. Vul, Ya. G. Sinaĭ, K. M. Khanin, *Feigenbaum universality and thermodynamic formalism*, Russian Math. Surveys **39** (1984), no. 3, 1-40.