A type for Taylor series for the C++ library iRRAM for exact real arithmetic

Florian Steinberg

last edited October 16, 2014

Contents

An introduction which can be skipped without guilty conscience

This text is meant to be a documentation of an expansion of the `C++` library `iRRAM` for exact real arithmetic by a type for real analytic functions. The guideline for explicitness is as follows: A person who is familiar with the the basic notions of real computability theory, and a moderately experienced `C++` programmer, should be able, when handed this document and its references, to reproduce (and in particular understand) all the additions done to `iRRAM` within a few days. This means we will not describe the added code line by line, but address the main obstacles that arose and explain how they were overcome.

For the time being, `iRRAM` is still in development. Thus some details may, and some hopefully will change. We will try to mark all parts of the text which seem likely to become irrelevant or incorrect in the future red. More general we colour all parts, that are temporary. Ideally these sections should vanish sometime.

The paper is divided into a number of parts. The first part serves to lay down the aspects of computability theory which are essential to the program but are not expected to be known by the reader. In the second part the we will describe some parts of `iRRAM` which can not be found in any of the documentations, but are essentially to what we want to do. The third part presents some parts of the `C++11` standard template library needed and then changes over to describes the key features and functionalities of the implementation. In the last chapter we will address some shortcomings and possible future improvements.

As we describe an implementation, we choose a very informal style of writing. No definitions, lemmatas and theorems even in the mathematical parts. Instead aim to make the text accessible by use of short chapters with descriptive names and many cross-references. Maybe we might add an index, lateron.

Part I
Higher type computability theory

We assume the reader is familiar with the concepts of computability of real numbers and functions thereon (at least functions on compact intervals). For look-up we suggest [PER89], or the freely available [BHW08] and the references given there. Here the former adopts a somewhat analytical point of view, while the later stays closer to the notions of computability theory. Another good source is probably [Wei00]. As the main source for things to implement, we feel like the paper [KMRZ12] should be mentioned in the introduction of this chapter. Tough it will be referenced at the multiple places where it is used.

In the later sections (which might be of lower importance for the implementations) we will discuss complexity theory of functions and operators. These topics are not thoroughly discussed in any of the above citations and we point the reader to [].

1 Analytic functions and computability

Let $B(x,r)$ denote the open disk of radius $r \in \mathbb{R}^+$ around some $x \in \mathbb{C}$ and $\overline{B(x,r)}$ its closure. The space of functions we are interested in is $C^\omega\big(\overline{B(0,1)}\big)$, that is the functions analytic on a open superset of the closed unit disc. We will abbreviate this space as C_1^ω. For the rest of this section we fix a function $f \in C_1^\omega$.

Since f is analytic of an open superset of the unit disc, it is uniquely determined by the series of its Taylor coefficients in zero, which has a radius of convergence strictly larger than 1. In the following, we will denote the series of Talyor coefficients of f in zero by $(a_n)_{n\in\mathbb{N}}$. It can be expressed in terms of the values of fs derivatives in zero, or by Cauchy's differentiation formula:

$$a_n = \frac{f^{(n)}(0)}{n!} = \frac{1}{2\pi i}\int_{|z|=r} \frac{f(z)}{|z|^{n+1}} d\lambda.$$

It is well known, that this series is again computable, if the analytic function f is (see for example [PER89, Chapter 1.2, Proposition 1]). Also the inverse is true: Given some series $\overline{b} = (b_n)_{n\in\mathbb{N}}$ whose radius of convergence R is strictly larger than 1, one can easily construct an algorithm computing the function

$$f_{\overline{b}} : B(0,R) \to \mathbb{C},\ x \mapsto \sum_{n\in\mathbb{N}} b_n x^n.$$

Thus the function f is computable if and only if its series of Taylor coefficients in zero is.

The above can be interpreted as statement about an operator from a suitable space of functions to a space of sequences and its inverse.1 We asserted, that those operators preserve computability point wise. Unfortunately this does not imply their computability: If the series $(b_n)_{n\in\mathbb{N}}$ is considered as input instead of being fixed, an algorithm computing the evaluation fails to exist. To construct the algorithms for a fixed series, we had to resort to information about it that can not be extracted from it in a computable way.

This is a very unpleasant fact: We want to represent analytic functions by their power series, since many manipulations can then be carried out efficiently (consider for example differentiation). But by the above we will not be able to provide an algorithm that evaluates an arbitrary function! Fortunately, this problem can be solved by manually enriching each function (or series) by a manageable set of additional Information. A more detailed presentation of this can be found in [Mül95], where also a set of sufficient information is specified.

We now proceed to discuss an adequate format for this additional information in our setting (which was already suggested in [KMRZ12]).

1.1 The constants k and A

Consider a sequence $\overline{a} = (a_n)_{n\in\mathbb{N}}$, such that the radius of convergence R of the sequence is strictly larger than 1 and let $f_{\overline{a}}$ denote the function

$$f_{\overline{a}} : B(0,R) \to \mathbb{C},\ x \mapsto \sum_{n\in\mathbb{N}} a_n x^n,$$

and f its restriction to $\overline{B(0,1)}$. We will always be able to find two positive integer constants k and A with the following properties:

- $r := \sqrt[k]{2}$ is strictly smaller than the radius of convergence R of $(a_n)_{n\in\mathbb{N}}$
- $|a_n|r^n \leq A$ for all $n \in \mathbb{N}$.

We briefly discuss how such constants can be found: Since the radius of convergence of $(a_n)_{n\in\mathbb{N}}$ is assumed to be strictly larger than 1, and $r_k := \sqrt[k]{2}$ goes to 1 as k goes to infinity, it is possible to choose k big enough for r_k to be in between 1 and the radius of convergence. Now fix such an k and abbreviate r_k as r. Consider the function

$$f_{\overline{a}}|_{\{z:|z|=r\}}.$$

Since this is a continuous function on a compact domain, it will be bounded. Let A be a bound of this function. The Cauchy differentiation formula assures, that

$$|a_n| = \left|\frac{1}{2\pi i}\int_{|z|=r}\frac{f(z)}{|z|^{n+1}}d\lambda\right| \leq \frac{A}{r^n},$$

thus A is as desired.

It is not hard to see, that there exists a machine that given a tripple $((a_n), k, A)$ and oracle access to approximations to some argument $z \in$ returns approximations to $f(z)$. We will elaborate on the runningtime of this algorithm in **??**.

1.2 A Lipschitz bound

Of course the zero-th tail estimate can be interpreted as a bound of $\|f\|_\infty$. Since any bound on the maximum norm of the derivative of a function is a Lipschitz constant of said function, this allows us to easily extract a Lipschitz constant: From the proof of Theorem 3.3 in [KMRZ12] can be seen, that the constants k' and A' for the derivative f' can be chosen as:

$$k' := 2k$$

and

$$A' := \left\lceil \frac{A}{r}\left(1 + \frac{2k}{e\ln(2)}\right)\right\rceil.$$

Together with the tail estimate, we conclude that

$$L := \left\lceil A \frac{\left(1 + \frac{2k}{e \ln(2)}\right)}{r - \sqrt{r}} \right\rceil$$

is a Lipschitz constant. Although the ceiling function is not computable, we can calculate a number which will either equal L or $L + 1$ with very little computational effort.

1.3 Choice of the constants k and A

It is obvious, that the constants k and A are not uniquely determined: They can always be increased and will still do. But if we fix either one of them, there is a smallest possible value for the other. For concrete functions, it is often the case, that multiple choices of the Parameters are available: Consider for example polynomials with coefficients from $[0, 1]$: Since these are whole functions, we can always choose $k = 1$. But if we do so, A will have to majorize the polynomial on the set $\{z \mid |z| = 2\}$, thus it will in general be of the order of 2^n, where n is the degree of the polynomial. On the other hand, if we choose k to be the degree of the polynomial, then A too will be of the order of the degree.

We can not give a decision procedure, but there is a guideline: In [KMRZ12] it is shown, that many operations on Analytic functions are computable in time polynomial in the output precision plus $k + \log(A)$ (we will come back to this in a little more detail in **??**). Thus it seems to be a good idea to optimize the asymptotic behavior of this parameter. However, this does not decide which of the two options given in the preceding example is to prefer: In both cases the parameter $k + \log(A)$ grows linearly with the degree of the Polynomial. In this case, the second possibility was chosen. This decision may be justified by the resulting memory consumption, but was mostly arbitrary.

1.4 Time bounds for the operations

1.4.1 Evaluation

The constants k and A enable us to estimate the size of the tail the infinite sum defining the function $f_{\overline{a}}$:

$$\left|\sum_{n>N} a_n z^n\right| \leq A\frac{(|z|/r)^{N+1}}{1-|z|/r}. \tag{1}$$

And, since r is strictly larger than 1, the right hand side will tend to zero if $|z|$ is smaller than one. This allows us to computably evaluate the function f (the restriction of $f_{\overline{a}}$ to $\overline{B(0,1)}$): First we insert $|z| \leq 1$ into eq. (1) and solve the resulting demand

$$\left|\sum_{n>N} a_n z^n\right| \leq \frac{A}{r^{N+1}-r^N} \overset{!}{\leq} 2^{-n-1}$$

for N. This resulsts in

$$N \geq \left(n + \mathrm{lb}\left(\frac{1}{1-2^{-\frac{1}{k}}}\right) + \mathrm{lb}(A) + 1\right)k + 1.$$

To bound the second term note, that the function $x \mapsto \mathrm{lb}(1+x)$ is convex, its value in zero is 0 and its derivative in zero is $\mathrm{lb}(e)$. Thus, for $-1 < x$ we have $\mathrm{lb}(1+x) \leq \mathrm{lb}(e)x$. Using this with $x = \frac{1}{\mathrm{lb}(e)k}$ we get

$$\frac{1}{2^{\frac{1}{k}}-1} \leq \frac{1}{2^{\mathrm{lb}\left(1+\frac{1}{\mathrm{lb}(e)k}\right)}-1} = \mathrm{lb}(e)k \leq 2k$$

and therefore

$$\mathrm{lb}\left(\frac{1}{1-2^{-\frac{1}{k}}}\right) = \mathrm{lb}\left(1+\frac{1}{2^{\frac{1}{k}}-1}\right) \leq \mathrm{lb}(1+2k).$$

We end up with

$$N = \mathrm{O}(k(n + \mathrm{lb}(k) + \mathrm{lb}(A))).$$

Next, we need to evaluate the Polynomial $\sum_{i=0}^{N} a_i x^i$ with precision greater than half the demanded. Of course, the time needed to do this will depend on the time needed to compute the coefficients a_i and the real number x. So let $T_{\mathbf{a}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ denote a time bound for the running time of a_i. That is: There exists a machine, that given i and n as input, produces a dyadic approximation of a_i with quality 2^{-n} and needs less than $T_{\mathbf{a}}(i,n)$ steps. Furthermore, assume that for $T_x : \mathbb{N} \to \mathbb{N}$ is a running time of x in the same sense. For simplicity we will always assume all running time bounds to be monotonous in all arguments.

Note the following: If T_x is a running time bound of x and T_y is a running time bound of y, then

- a running time bound for $x + y$ is given by

$$T_{x+y}(n) := T_x(n+1) + T_y(n+1) + O(n)$$

  where the last term comes from adding the approximations.

- assuming $x, y \leq 1$, a running time bound for xy is given by

$$T_{xy}(n) := T_x(n+1) + T_y(n+1) + O(n \log(n) \log(\log(n)))$$

  where the last term comes from multiplying the approximations.

By using Horner's Algorithm one can now obtain an running time bound for the evaluation of a polynomial of degree N. Computing the required approximations will take time

$$T_{app}(n) = \sum_{i \leq N} T_a(i, n+2i+1) + T_x(N+1) \leq (N+1)T_a(N, n+2N+1) + T_x(N+1).$$

Computing an approximation to the function value from the obtained approximations will take N multiplications and N additions, therefore we obtain for the time T_{cmp} needed to do this

$$T_{cmp}(n) = O(N^2 \log(N) \log(\log(N))).$$

This leaves us with the following time bound for evaluation of the function f:

$$T_f(T_{\mathbf{a}}, k, A, T_x, n) = O(N^2 \log(N) \log(\log(N))) + T_a(N, n+2N+1) + T_x(N+1)).$$

If we furthermore assume that $T_x = \mathrm{O}(n)$ and $T_a = \mathrm{O}i + n$ we get

$$T_f(k, A, n) = \mathrm{O}(...)$$

1.4.2 Evaluation of derivatives

From [KMRZ12] it is known, that for the d-th derivative $f^{(d)}$ of the function, we can use the constants $k_d := 2k$, $A_d := A2^{-\frac{d}{k}} d^d (1 + \mathrm{lb}(e)k)^d$ to accompany the derived series

$$a_i^{(d)} := \prod_{j=1}^{d} (j+i) a_{i+d}.$$

To find a running time bound for the evaluation of the derivative, we can combine the results from the previous section with the new constants and the updated running time of the sequence. A running time bound for the derived sequence is given by

$$T_{\mathbf{a}^{(d)}}(i, n) = T_{\mathbf{a}}(i + d, d \cdot \mathrm{lb}(i+d)) + \mathrm{O}(d(d+i) \log(d+i) \log(\log(d+i)))$$

Considering the new k and A gives a new value N_d for the needed number of coefficients:

$$N_d \geq \left(n + \mathrm{lb}(1+4k) + \mathrm{lb}(A) - \frac{d}{k} + d\mathrm{lb}(d) + d\mathrm{lb}(1+\mathrm{lb}(e)k)\right) 2k + 1,$$

thus

$$N_d = \mathrm{O}\left((n + \mathrm{lb}(A) + d\mathrm{lb}(k) + d\mathrm{lb}(d))\, k\right).$$

This and the new running time of the series together with the formula from the previous chapter leads to the bound

$$O(N_d^2 \log(N_d) \log(\log(N_d)) + T_a(N_d + d, n + 2N_d + 1) + T_x(N_d + 1))$$

for $T_{f^{(d)}}(T_{\mathbf{a}}, k, A, T_x, n)$

1.4.3 The Taylor sequence around another point

Computing the d-th coefficient of the Taylor-sequence of f around a point x can be done by evaluating the d-th derivative and dividing by the faculty of d. Since computing the faculty of d takes d multiplications of numbers of size smaller than d, the time needed can be obtained from the previous chapter by adding another term

2 Real computability

The next two sections introduce some theoretical background. It is inevitable that there is some repetition of material the reader is already familiar with. For the first reading we advise, that these chapters should be skipped and be reverted to, when they are needed (if this is the case at all). There will be cross references to make this bearable.

At first sight, it might seem strange, that such an excessive amount of theory is needed for an implementation which seems straight forward from the content of the preceding section. But there are two reasons we chose to take such an extent of theory into consideration anyway: Firstly it should be possible to extend this code to more general constructions, for example Analytic functions on closed subsets Intervals or even the Gevrey-classes discussed in the later chapters of [KMRZ12], and secondly aligning the code according to the theory turned out make orientation a lot easier.

Of course there are many other notions of complexity, which do not turn out to be equivalent to those introduced above. Some examples can be found in **??**

2.1 Representations, a reprise

Recall that the space $\mathcal{C} := \{0,1\}^{\mathbb{N}} \subseteq \mathbb{N}^{\mathbb{N}}$ is called cantor space and that an element $w = (w_1, w_2, \ldots) \in \mathcal{C}$ is called computable, if there is a Turing-machine returning w_n upon input of n (that is, if w is computable as function from $\mathbb{N}$ to $\mathbb{N}$). Furthermore a function $f : \mathcal{C} \to \mathcal{C}$ is called computable, if there is a oracle Turing-machine, returning $f(w)_n$ upon input of n, if granted oracle access to w (as function).

Recall that a representation ρ of a set X is a partial surjective mapping from cantor space to X. In this case, given $x \in X$, the elements of $\rho^{-1}(x)$ are called *names* of x. Given representations ρ_i of sets X_i ($i \in \mathbb{N}$) it is possible to construct a representation $\prod_{i\in\mathbb{N}} \rho_i$ of $\prod_{i\in\mathbb{N}} X_i$ (using some computable, bijective function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$), and also a representation $\rho_1 \times \rho_2$ of $X_1 \times X_2$ (by interleaving the names). If ρ is a representation of a set X the pair (X, ρ) is called a represented space and sometimes simply denoted by X, an element x of a represented space (X, ρ) is called *ρ-computable*, if it has a computable name.

If (Y, ν) is another represented space, a function $f : X \to Y$ is called $\rho \to \nu$-computable, if there is a computable realizer, that is a computable function $F : \mathcal{C} \to \mathcal{C}$ such that the following diagram is commutative:

$$\begin{array}{ccc} X & \xrightarrow{f} & Y \\ {\scriptstyle\rho}\uparrow & & \uparrow{\scriptstyle\nu} \\ \mathcal{C} & \xrightarrow[F]{} & \mathcal{C} \end{array} .$$

It is well known, that if topologies are given on X and Y, and if the representations are well adjusted with respect to these topology, this notion of computability implies continuity.

When represented spaces X and Y are given, it is possible to specify a representation of $C(X,Y)$ that renders exactly the computable functions computable. For $X = [0,1]$ another representation exists: One can represent a continuous function by (a suitable encoding of) a sequence of Weierstraß-polynomials approximating the function uniformly. That this representation also leads to the same notion of computability is not obvious but true.

2.2 Standard representations

The standard representation ρ_{dy} of $\mathbb{R}$ is defined as follows: $\rho_{\mathrm{dy}}(w) = x$ if and only if w encodes a sequence z_n of integers, such that $\left|x - \frac{z_n}{2^{n+1}}\right| \leq 2^n$. The standard representation of $\mathbb{C} \cong \mathbb{R}^2$ is $\rho^2_{\mathrm{dy}} = \rho_{\mathrm{dy}} \times \rho_{\mathrm{dy}}$. ρ_{dy}-computability coincides with the standard notion of computability of real numbers (the same is true for complex numbers).

We can get a representation $\rho^{2\omega}_{\mathrm{dy}}$ of our space C^ω_1, by naming f the same as the sequence of its Taylor coefficients $(a_n)_{n\in\mathbb{N}} \in \mathbb{C}^\omega = \prod_{i\in\mathbb{N}} \mathbb{C}$. Now we can reword the comments from the beginning of section 1: The function

$$\mathrm{ev} : C^\omega_1 \times \mathbb{C} \to \mathbb{C},\ (f,x) \mapsto f(x)$$

is not $\rho^{2\omega}_{\mathrm{dy}} \times \rho^2_{\mathrm{dy}} \to \rho^2_{\mathrm{dy}}$-computable. We can fix this by considering a slightly adjusted representation π, that will precede a $\rho^{2\omega}_{\mathrm{dy}}$ name of a function f by two integers k and A as described in section 1.1 above (we skipped some technical details here. For example k should be encoded in unary and A in binary). This adaptation renders evaluation computable.

2.3 Complexity of reals

Given a representation ρ of a set X, the following notion of complexity suggests itself: Call an Element x of X computable in time $t : \mathbb{N} \to \mathbb{N}$, if there exists a Turing-machine computing some name of x in this time. This approach has the advantage of working in the most general setting, but for more concrete representations there often is a more natural notion of complexity.

For example consider the the standard representation ρ_{dy} of the real numbers: One could also define the complexity of x by using Turing-machines M_{dy} returning (encodings of) the Integers z_n such that $|x - \frac{z_n}{2^{n+1}}| \leq 2^n$, instead of the digits of their encoding. It is not difficult to see, that one can obtain a Turing-machine of the former kind from one of the latter without increasing the running time by more than a polynomial factor and vice versa. Thus both procedures fortunately lead to the same class of polynomial time computable real numbers.

3 Higher type complexity theory

Since we introduced computable functions as being realized by oracle Turing-machines, we need a measure of complexity for those machines. Recall that the running time of an oracle Turing-machine with a fixed oracle is defined very similar to that of an regular Turing-machine, where a query to the oracle is counted as one step (while the writing of the input for the oracle and the reading of the output will commit to the running time). The oracle Turing-machine is then said to have complexity bounded by some function $t : \mathbb{N} \to \mathbb{N}$, if the running time is bounded by this function for any fixed oracle.

3.1 Complexity of functions

Let ρ and ν be representations of some sets X and Y. The complexity of a function $f : X \to Y$ is said to be bounded by some function $t : \mathbb{N} \to \mathbb{N}$, if there is a realizer F whose complexity is bounded by this function. A function is said to be computable within polynomial time, if its complexity can be bounded by a polynomial.

Though this definition is well established, we add some remarks. First of: for functions $f : \mathbb{R} \to \mathbb{R}$, it might be natural to use oracle Turing-machines returning integer values and accepting integer valued oracles. Similar to the discussion section 2.3 one can conclude that this will not alter the class of polynomial time computable functions.

Still this definition is to be taken with a grain of salt: the following might not be the best choice of an example, maybe a better one involving the multiplication of reals can be found in Norbert Müllers Draft? Consider a function $f : \mathbb{R} \to \mathbb{R},\ x \mapsto a$, for some number $a \in \mathbb{R}$ computable linear time, but not in constant time (according to a fixed kind of Turing machine representing reals). We want to compare two algorithms computing this function. For this let P be an an algorithm computing a in linear time. Now consider for once the algorithm doing one step of P, then counting up to the total steps done in P up to this point in time and returns what P returns if P finishes. On the other hand the algorithm which does one step of the algorithm P and then queries the oracle with the number of steps done in P and in the end returns the result of P. These algorithms are considered to have the same running time according to the definitions above. However, it should be clear that the former is clearly superior in practice, where the oracle will be some algorithm taking time to evaluate.

To capture this difference we say that a function $f : X \to Y$ is computable within time $t : \mathbb{N} \to \mathbb{N}$ and with lookahead $s : \mathbb{N} \to \mathbb{N}$, if it is computable in time t while, upon input of n, not querying the oracle on numbers larger than $s(n)$. In the example above, the first algorithm is computable within linear time and with constant lookahead, while the second is linear time with linear lookahead. Of course there are more elaborate examples, where one really has a tradeoff between the running time and the lookahead.

As writing large numbers on the query tape will take steps of computation,

any polynomial time computable function can also be computed with polynomial lookahead. This implies that polynomial time computable functions map polynomial time computable reals to polynomial time computable reals.

3.2 Complexity of operators

The refinement of the representation $\rho_{\mathrm{dy}}^{2\omega}$ to the representation π discussed in section 2.2 does not only render the evaluation computable, but will also make a lot of other operations computable in polynomial time. A more elaborate presentation of this can be found in [KMRZ12] (where also the term polytime is made explicit). In particular in Theorem 3.3 of [KMRZ12] the following operations on C_1^ω are shown to be polytime computable with respect to the adjusted representation:

a) Evaluation as discussed above.

b),c) Addition and multiplication.

d),e) Differentiation and anti-differentiation.

f) Parametric maximization. This needs to be explained a little closer.

The main effort is to work out bounds for the constants k and A of the new functions. This requires multiple technical Lemmata.

Part II
iRRAM

In this part, we are going to describe some key features of the C++ library iRRAM for exact real arithmetic. We do not claim our depiction to be complete or self-contained. Instead we restrict ourselves to those parts of iRRAM that are of importance for us, and can neither be found in the official iRRAM documentation [Mülb], nor in the incomplete html-documentation [Müla] or the draft [Mül13] (yet?).

4 Basic functionalities

Following the theory it might be expected that the proceeding depicted in sections 2.2 and 2.3 is the most reasonable approach to implementations of real analysis. In this spirit, a real number x should be represented by an algorithm $M_{\mathrm{dy}}(x)$, taking a natural number n and returning an approximation, i.e. an appropriately encoded rational (or dyadic) number x_n such that $|x - x_n| < 2^{-n}$. It turns out that this proceeding has major drawbacks:

(P1) Each time a sum or a product of real numbers is calculated, the algorithms of the corresponding real numbers must be copied or at least referenced. This leads to a tree-like structure (more precisely a directed acyclic graph) of any program and to extensive memory consumption of some standard operations like computing the powers of some real matrix or summing up power series.

(P2) An algorithm P encoding a real number carries more information than the real number itself. If P is given to a function, this function can for example also use the running time of P to generate its output. This kind of functions should from a mathematical point of view not be considered computable, and can lead to pathological behavior.

It might be suspected that (P1) is not significant, since the memory consumption is caused by a high number of pointers, which are tiny. However, experience shows that this does cause major problems in applications that arise in practice.

4.1 `iRRAM`: another approach

Thus the `iRRAM` chooses a different approach. Namely, it will choose a fixed precision and calculate a finite interval of size smaller than this precision from each Algorithm representing a real number. Then all the manipulations on real numbers will be carried out on these intervals instead. If a situation is encountered, where the size of these intervals is not sufficiently small anymore (for example two reals are supposed to be compared and the intervals overlap), the whole computation is restarted with a higher starting precision. The procedure of restarting the calculation is called a *reiteration*. The single runs of the program with different precisions will be referred to as *iterations*.

We hint at why this advance might not suffer the problems mentioned in the introduction of this section: A major part of (P2) was that the running time of an oracle is accessible (for instance by comparing time stamps before and after the query). Since any information about earlier iterations is lost, the running time of an algorithm computing a real number should not be easily accessible (of course the power of the programming language `C++` makes it very difficult to assure this). The main point of (P1) was the dependence of sums and products of reals on the algorithms of the summands or factors. Since reals are represented by finite intervals, and the interval representing the sum of two

reals can be saved without reference to the intervals representing the summands, this problem ceases to exist.

At first glance, this approach seems to be very time consuming: The whole computation might be restarted repeatedly, discarding all the computations made in the earlier iterations completely. But it is well known, that this does not blow up the asymptotic complexity: More precisely, the asymptotic complexity of the whole computation and the last reiteration coincide [].

Nontheless this approach has some real (but hopefully more manageable) drawbacks

- If the program includes in and output, a restart of the program will lead to doubled output. This is mainly a implementation problem and can be avoided by using the output methods provided by `iRRAM`.
- If on the other hand, the program asked the user for input at some point, this input will have to be memorized. Moreover: if any multivalued function is computed, it has to evaluate to the exact same value in the next run and has to be memorized. This is to avoid incoherences in output.
- Each time a reiteration is triggered, the whole program restarts. This means, that nearly everything (as mentioned above some things are memorized) is re-evaluated. In particular: If the program is composed of two tasks, and one of those needs a higher precision, both tasks will be carried out in this higher precision. This means, that reiterations are very expensive in terms of computation time.

The problem described in the last bullet can partially be eliminated by dividing the tasks into two sub-programs. However, as mentioned above, this does not effect the asymptotic running time, and we thus will not go into it in more detail. Instead we will go into the problem discussed in the first bullet in more detail in the oncoming section. Note, that the content of the remaining bullet is very closely related. It may or may not be addressed on its own.

Another point is, that programs for `iRRAM` are written in `C++`. Since `C++` is a very powerful programming language. This means, that it is often possible for the user to do things he is not really supposed to do.

4.2 Output

We did already mention, that output can be a problem and that one should use the output methods provided. `iRRAM` offers the following functions for output:

- The functions `iRRAM::rwrite`, `iRRAM::rwritee`.
- The function `iRRAM::rfprint`.
- The function `iRRAM::rshow`.
- Any `iRRAM::orstream` via the overloaded `<<` operators. In particular the standard one: `iRRAM::cout`.

A description of how the output functions are to be used can be found in [Mül13, Chapter 34.8.].

We emphasize once more: It is important to acknowledge the restrictions of the `orstream`s in comparision to `ostream`s: For example pointer adresses will not be displayed. This makes perfect sense, since adresses may vary throughout the iterations! Output of the same pointers adress might thus result in different adresses at different points of the computation. In this case the `iRRAM` takes care of the problem by not outputting pointer addresses but first casting them to `bool`s. A real example: If one outputs the error of some `REAL` it might show values greater than one, which might not reflect the future behaviour as the very next command could be to output the REAL, which will then trigger a reiteration and increase precision. Since the output will not be revised the result can be confusing for the user. Although these problems should be handled by `iRRAM` itself, it is clear that there will always be some ways to trick the system and it is advised to handle output with care.

5 Data types

5.1 REALs and sizetypes

The data type `REAL` is the heart of `iRRAM`. An overview over the main functionalities can be found in [Mül13]. We will take a closer look at the implementation. As already mentioned, real numbers will be represented by intervals of finite size in each iteration. To achieve the best possible results at low precision, `iRRAM` will work with pairs of the usual floating point type `double` in the first iteration. We will omit this part and concentrate on the parts that are used in the later iterations:

value: A pointer to a MP object which will be interpreted as the centre of the interval.

error: An object of the `iRRAM` internal type `sizetype` that will be interpreted as the distance of the centre to each of the endpoints of the interval.

The standard operators `+`, `*`, `-` etc. are defined for Real numbers. They do interval arithmetic, which boils down to error propagation.

5.2 INTEGERs

As the name might indicate, the class `INTEGER` is a class to model integers. The main advantage over the `C++` type `int`,is that there is no restriction on the size of the Integer. The main drawbacks are size and speed. Thus it should only be used, when an unbounded size is really necessary.

Two examples: We will use the data type `INTEGER` for the constants k and A described in section 1.1, since a reasonable small number of iterated differentiations can lead to a growth of these constants which would lead to an overflow of `int`s (or `unsigned int`s for that matter). On the other hand, we will model series as maps that map `unsigend int`s to `REAL`s, since it seems to be very unlikely that we will ever need to access a element of the sequence beyond the 4.294.967.295-th. Especially since we are mostly dealing with effectively convergent series. This does contradict the purpose of `iRRAM` as library for *exact* real Arithmetic, as it leads to the existence of a best possible approximation and the possibility of an overflow (independent of the available memory). It might be necessary and would be interesting to investigate the severity of this limitation. Also it is not clear to the author whether similar restrictions do already exist in other parts of `iRRAM`. If so it would be interesting to compare them.

??

describe how INTEGER and REAL interact

5.3 The class COMPLEX

The complex numbers play a very important role in the theory of the analytic functions. This importance will only be partially reflected in our implementation. The reason is that equality is not computable. This means that it can not

be checked whether the imaginary part of a complex number vanishes. This entails the following two drawbacks when handling real valued analytic functions as a special case of analytic functions with complex coefficients:

- Since we can not detect whether a sequence is really real valued, all operations would have to be carried out with the complex series with vanishing imaginary part. This means we would spend a lot of time on unnecessary operations of adding and multiplying zeros.
- For similar reasons, we would have to output complex numbers upon real input. This can not easily be fixed, since there is no canonical way to convert a complex number to a real (compare **??**).

We will thus work with real coefficient series, and remark that these can be easily used to implement analytic functions with complex coefficients, since the real and imaginary part are analytic functions with real coefficients.

Nonetheless we will want to allow evaluation of real analytic functions in complex arguments, thus we need a more or less complete type for complex numbers. `iRRAM` provides the class `COMPLEX` for this purpose. As we do already have a type for real numbers, the implementation of this type is very simple and need not be further addressed here. However, the type lacks some basic capabilities we will need. For example, the operators `<<` and `+=` are not overloaded and there are no constructors from types other than `REAL`. We give a short description of the improvements we made:

constructors: We added the following three constructor templates:

```
template<class T>
COMPLEX(const T& real_part);
template<class T, class S>
COMPLEX(const T& real_part, const S& imag_part);
template<class T, class S>
COMPLEX(std::pair<T,S> p);
```

The first takes an arbitrary type `T`, attempts to convert this type to a `REAL`, and then constructs a `COMPLEX` having the result as real part and imaginary part zero. The second does the same, but for both real and imaginary part. The third constructor will accept a `std::pair` instead of two parameters. This is in particular useful, since it enables the use of nested lists to construct vectors of complex numbers:

```
vector<COMPLEX> v = {{1,2}, {pi(),4.5}};
```

operators and functions: The operators `+=` and `*=` where implemented in the obvious way. Furthermore the two functions

```
COMPLEX power(const COMPLEX&, const COMPLEX&);
COMPLEX power(const COMPLEX&, const int);
```

were added. The implementations are slightly adjusted copies of those of the real counterparts.

output: The operator `<<` was overloaded. We thereby followed the conventions:

- The precision is to be interpreted point wise. This means, both the real and the imaginary part are to be printed with the specified precision. This corresponds to the use of the supremum norm on $\mathbb{C} = \mathbb{R}^2$.
- Complex numbers are to be coated in parentheses. This is to avoid confusion. For example the attempt to output a linear monomial via a template as follows:

```
template<class T>
void out(const T& x) {
  cout << x << " * X" << endl;
}
```

  would otherwise lead to ambiguous (or wrong) output.

- If the imaginary part of the number is smaller than the desired precision, only the real part is to be printed (i.e. `+1.00E+0001` instead of (`+1.00E+0001+ 0 i`)). The same for small real part. Of course, if both real and imaginary part are small, the real part is to be printed.

The other standard output operators, in particular `rwrite` were also overloaded.

6 iRRAM functions

6.1 Limits

6.2 iRRAM::FUNCTIONs

We aim to implement a type for analytic functions in iRRAM, which is not yet present. But there already is a type of functions implemented. We will give a short review of this type.

Part III
Implementation

It should be clear, that the right place for our implementations would be the file iRRAM_functional.h (and maybe a corresponding .cc file). As `iRRAM` does not yet use `C++11`, which our implementations will rely on heavily, we temporarily locate it as separate file called FUNC.cc in the folder 'user_programs' (or in more current, not yet released versions which miss this folder, in the folder 'examples').

7 Tools provided by the C++11 standard template library

In this section we will introduce those tools the C++11 standard template library provides, which we will use extensively. The information presented was gathered from various web pages. Some of the most frequented sites were [cpp] and [cpl], many of the discussions on the site [sta] were very helpful. Last but not least, Carsten Rösnick was of great help by hinting at what to look for.

C++ provides four kinds of functional objects: functions, member functions, function pointer and member function pointer. Since the first two are very common, we assume the reader is familiar with them. However, it is important to note that a function pointer can only point at functions that where created outside of the main program. This makes it impossible to dynamically create new functions when needed, which is a serious shortcoming for us, since we want to be able to add and multiply functions.

Member function pointer do actually solve this problem, since classes, and with them their member functions, can be dynamically created and destroyed. But using member functions to achieve the flexibility we desire would be very involved, luckily for us C++11 added improved syntax for exactly this. In the following sections review these tools.

7.1 std::functions

The std::function template is a general-purpose polymorphic function wrapper (according to [cpp]). We will find `std::functions` to be highly useful. But it will be not until we learn about the C++ lambda calculus, that we can grasp their whole potential. Thus the description in this chapter might appear somewhat unspectacular. A `std::function` can be defined by

```
1 std::function<RESULT(PARAM)> f;
```

A std::function can be evaluated like a function, and defined from a function pointer, as the following short example shows:

```
1  #include <iostream>
2  #include <functional>
3
4  using std::function;
5
6  double f(int i) {
7    return double(i);
8  }
9
10 int main()
11 {
12   function<double(int)> g = f;
13   std::cout << g(4) << std::endl;
14   return 0;
15 }
```

The `std::function` can do a lot more than regular functions though. For example: If `f` is a `std::function<RESULT(PARAM1, PARAM2)>` of two arguments, we can define a `std::function<RESULT(PARAM2)> g` by setting the first parameter to a fixed value (say `x`) using the function `std::bind`:

```
function<RESULT(PARAM2)> g = bind(f, x, std::placeholders::_1);
```

To achieve something similar with regular function pointers is complicated (though it is possible). We will see that some of those possibilities hold risks, since `iRRAM` might not expect to encounter a function, containing real numbers that are not handed to it as an parameter.

The syntax `function<RESULT(PARAM)>` of `std::function`s seems nicer than the `FUNCTION<PARAM, RESULT>` we have seen before. It is also more convenient, since confusing parameter and result type gets a lot harder. The former syntax is implemented by a template specialisation. The definition of looks like this:

```
template<class T>
class function {};

template<class T, class S>
class function<T(S)> {/*...*/};
```

Here the first two lines define an empty template class, then lines four and five specialize the definition in the case, that the template argument is of a specific form. Namely a string of the form T(S), where T and S are some arbitrary types. We will use this trick to also improve the syntax of our function type.

`std::function`s are in many respects superior to `C++` functions, but they lack one thing functions do have: the possibility to be made templates.

7.2 The C++ lambda calculus

One of the main sources for `std::functions` is the `C++` lambda calculus. A lambda expression in `C++` is of the form

```
[/*captures*/](/*parameters*/) -> /*output_type*/ {/*algorithm*/};
```

Where the commented parts are to be replaced as follows:

`/*captures*/` is to be replaced by a list of variables of local scope, that are needed by the algorithm but supposed to appear as parameters of the function (those are actually what mathematicians mean, when they say 'parameters'). A variable occurring in this list will be called captured (by the lambda function). Global variables need not be captured, since thy can be accessed anyway. Variables may be captured by copy or, if they are preceded by an '`&`', by reference.

`/*parameters*/` is to be replaced by the list of parameters of the function to be constructed.

`/*output_type*/` is to be replaced by the output type of the function to be constructed (if the value that follows the return command is not of this

type, a implicit type conversion will be attempted). This part (together with the '`->`') can be omitted if the output type will be clear from the context.

`/*algorithm*/` is to be replaced by the algorithm calculating the return value from the parameters and the captured variables.

an easy example of a definition of a `std::function` via the `C++` lambda calculus could look like this:

```
int i = 5;
std::function f<double(int)> = [i](const int& n) {i*log(n);};
```

Here the output type was omitted, since it is specified in the `std::function`.

We remark, that members can not be captured directly: if `c` is an object of an class `C`, which has a member k of type T, then

```
[c.k]() -> T {return c.k;};
```

will not work. The same is true if we for example try to capture `k` in the scope of the class definition.

The reason is as follows: Assume the class C has some member function f changing the value of k. In this case

```
[&c, c.k]() -> T {c.f(); return c.k;};
```

would be ambiguous, since c.k could mean both the field of the object c captured by reference, which has the new value, or the member c.k, which was captured by copy before the change was made and therefore should have the old value. The same ambiguousness arises, when a member is captured in the scope of the class definition, since the `this` pointer may also be captured.

Thus we will have to first copy the member to a local variable, then capture it.

7.3 `std::shared_ptrs`

A shared pointer is an object consisting of an pointer (to an object of specified type) and a pointer to an reference counter. Each time the shared pointer is copied the reference counter will be increased. If a shared pointer is destroyed, it decreases the reference counter and checks if it is zero, and if it is, it destroys the object it is pointing to.

Shared pointers are very useful for treelike ownership relations: If one object is used by multiple other objects, each of the latter can, instead of an pointer or a copy, to the former hold a shared pointer. This way it is guaranteed, that we neither have useless copies, nor memory leaks because no one feels responsible for destroying.

The `std::shared_ptr` can be dereferenced by a preceding $*$, just as a regular pointer, can be constructed from regular pointers and compared to the `NULL` pointer. Here is an short example:

```
double* ptr = new double(4.5);
std::shared_ptr<double> s_ptr(ptr);
cout << *s_ptr << endl;
```

returns '4.5'.

There are two main sources of errors when handling shared pointers:

- Multiple shared pointers are constructed from the same pointer: In this case each shared pointer keeps its own reference counter. If the first of those counters hits zero, the object will be destroyed. If now a shared pointer following an other reference counter tries to access the object, there will be an error.

- Circular ownership relations: for simplicity lets have a look at the case of two lonely shared pointers pointing at each other. Each of the pointers has a reference counter which is one (since they are lonely). now assume we destroy one of them. This will decrease the corresponding reference counter and, since the reference count will hit zero, destroy the other shared pointer. This in turn will decrease its reference counter, which will also hit zero. Therefore it will attempt to destroy the first shared pointer. Which will lead to an error, since it has already been deleted.

To avoid the first of these mistakes and memory leaks, we will try to use new exclusively in the initialisation of `shared_ptr`s.

8 A class for Taylor series

introduce abbreviations.

8.1 The class definition

```
template<class ARG>
class BASE_ANAL: public FUNC<ARG(ARG)> {
  // members:
    private:
      // the parameters which determine the function:
          ps_ptr coeff;
          INTEGER k;
          INTEGER A;
      // additional data, stored to speed things up.
          REAL effective_radius_of_convergence;
      bool has_better_alg = false;
      poly_ptr poly = NULL;
      INTEGER lipschitz;
      // alg_func_ptr<PARAM, RESULT> algorithm = NULL;
  // constructors:
    public:
      BASE_ANAL();
      BASE_ANAL(REAL z);
      BASE_ANAL(
      const ps& _coeff,
      const INTEGER& _k,
      const INTEGER& _A
    );
    BASE_ANAL(const POLY<REAL>&);
  // member functions:
    public:
      POLY<REAL> cut_of_at(const unsigned int&) const;
      ARG operator () (const ARG&) const;
      virtual void derive(unsigned int);
      virtual void anti_derive(unsigned int);
      void has_algorithm(const alg_func<ARG, ARG>&);
      void restore_alg();
      void show_info() const;
    protected:
      virtual void print(iRRAM::orstream&) const;
      virtual void print(int precision) const;
  // standard operators:
    public:
      BASE_ANAL<ARG>& operator = (const BASE_ANAL<ARG>& f);
      friend BASE_ANAL<ARG>& operator += (const BASE_ANAL<ARG>&);
      ...
      friend BASE_ANAL<ARG> operator *<> (
        const BASE_ANAL<ARG>&,
        constBASE_ANAL<ARG>&
      );
      template<class PAR>
      BASE_ANAL<ARG> operator () (const BASE_ANAL<ARG>&);
    // friends classes:
      template<class>
      friend class POLY;
```

```
51 };
```

8.2 The evaluation

9 Some additions

9.1 Output

9.2 Additional constructors

9.3 The class `coeff_fetcher`

The class `coeff_fetcher` is a helper class to enable caching of the coefficient values in `POWERSERIES` that also works if the `POWERSERIES` is declared a constant. Since the class `coeff_fetcher` is only a helper class and should not be available to the user and is put in a anonymous namespace.

To explain the use of the class `coeff_fetcher`, consider the product `P*Q` of two power series `P` and `Q`. Each time we call `(P*Q).get_coeff(n)`, the convolution of the power series is calculated, which involves `n` multiplications and `n` additions. There are cases, where the same coefficient is needed multiple times, for example if we want to evaluate the powers P^i of some power series. In this case it would be more efficient, to remember the coefficients and not to recompute them each time.

The above can be implemented by saving a vector `known_coeffs` of pointers to the coefficient type: Each time some, say the `n`-th, coefficient is requested, the function `get_coeff` will check, whether `known_coeffs[n]` is defined and not the `NULL` pointer. If it is, it will return the value pointed to. If it is not, it will calculate the value and set `known_coeffs[n]` to point to a coefficient type of the result value.

Thus far, this could be implemented in the class `POWERSERIES` itself. There is one problem: It is often reasonable to work with constant power series (for example the power series underlying a `BASE_ANALYTIC` should be constant for the exact same reasons the algorithm of a `FUNC` and the sequence of a `POWERSERIES` is). But in this case, the vector `known_coeffs`, as a member of a constant object, would also be constant. To avoid this, the vector will be stored in the external class `coeff_fetcher`, and be referenced in `POWERSERIES` only by pointer.

The declaration of the class realizing this looks as follows:

```
template<class c_t>
class coeff_fetcher {
  // members:
    private:
      vector<c_t*> known_coeffs;
      POWERSERIES<c_t>* powerseries;
  // con- and destructors:
    public:
      coeff_fetcher(POWERSERIES<c_t>*);
      ~coeff_fetcher();
  // member functions:
    public:
      c_t get_coeff(unsigned int n);
      void reset();
  };
```

We briefly discuss the components:

members: The member `known_coeffs` will, as discussed before, hold pointers to the already evaluated coefficients. Of course the `coeff_fetcher` will have to know, which power series to get the values from. Thus there is an additional member: A pointer to the said power series.

con- and destructors: The only constructor will construct a `coeff_fetcher` to a power series. The location of said power series is handed to the constructor as a pointer. Since the vector `known_coeffs` does hold pointers which will have to be deleted we need to implement a destructor. This destructor will simply call the function `restore` described below.

member functions: Of course there is a member function `get_coeff` which fetches a coefficient. This is done as depicted above. There is another member function `reset` that resets the vector `known_coeffs` to an empty vector. Therefore it will first delete all the contained pointers and then reallocate the vector. This is for the case that retaining the coefficients leads to intolerable memory consumption.

The implementations are not noteworthy.

Also we need to add some lines to the class declaration of `POWERSERIES<c_t>`. Most importantly:

```
coeff_fetcher<c_t>* get_coeffs = new coeff_fetcher<c_t>(this);
```

which creates, upon construction of a power series, a appendant `coeff_fetcher`. In the member function `get_coeff(n)`, we will then replace the call `(*coeff)[n]` by `get_coeffs.get_coeff(n)`. Secondly, `coeff_fetcher` will have to be declared a friend of `POWERSERIES` in order to be able to access the private member `coeff`. And finally, the class `POWERSERIES` will need an explicit destructor which deletes the associated `coeff_fetcher`.

We end the section with a remark concerning a possible issue with the constructors. A copy constructor is a constructor of type `coeff_fetcher(const coeff_fetcher<coeff_type>&)`. Since this constructor is essential for basic manipulations (for example initialisations) a default copy constructor will be declared by the compiler, if none is declared by the programmer. This default copy constructor will simply copy the object, and will cause problems if used: For assume that a power series is declared and initialized from an existing one

```
POWERSERIES<REAL> exp_pwr([](const unsigned int& n) -> REAL {
  return 1/REAL(faculty(n));
});
POWERSERIES<REAL> P = exp_pwr;
```

Since the default copy constructor will simply copy any member, `P` will also hold a pointer to the `coeff_fetcher` which `exp_pwr` uses. This `coeff_fetcher` is of course oblivious of this fact and will continue to read `exp_pwr`s coefficients. This means, that changes in `P` will not result in changes in the returned coefficients, and will even cause an error if `exp_pwr` is destroyed. One might assume, that this problem is solved by the template for a copy constructor we introduced in **??**, but it is not: a template will not be accepted as a definition of a copy

constructor. The compiler will deliver a default copy constructor, and since this default copy constructor is a better fit for the type than the template, it will also be used. To resolve this, we have to add another constructor.

9.4 Possible future extensions

It is clear, that this whole chapter should be written in red. For readability we restrain the colouration to the heading.

References

[BHW08] Vasco Brattka, Peter Hertling, and Klaus Weihrauch, *A tutorial on computable analysis*, New computational paradigms, Springer, New York, 2008, pp. 425–491. MR 2762094 (2012e:03142)

[cpl] *C++ reference*, http://www.cplusplus.com/.

[cpp] *C/c++ reference*, http://en.cppreference.com/.

[KMRZ12] Akitoshi Kawamura, Norbert Th. Müller, Carsten Rösnick, and Martin Ziegler, *Parameterized uniform complexity in numerics: from smooth to analytic, from np-hard to polytime*, pre-print (2012).

[Müla] Norbert Th. Müller, *iRRAM: exact real arithmetic*, As included as "irram.html" in the folder "doc" in the Version "2008_01" of `iRRAM`, released on http://irram.uni-trier.de/.

[Mülb] ———, *The iRRAM: Exact Arithmetic in C++*, As included as "irram.ps" in the folder "doc" in the Version "2008_01" of `iRRAM`, released on http://irram.uni-trier.de/.

[Mül95] ———, *Constructive aspects of analytic functions*, Computability and Complexity in Analysis (Ker-I Ko and Klaus Weihrauch, eds.), Informatik Berichte, vol. 190, FernUniversität Hagen, September 1995, CCA Workshop, Hagen, August 19–20, 1995, pp. 105–114.

[Mül13] ———, *iRRAM*, obtained from the author directly via email. Not available to anyone else, yet, 2013.

[PER89] Marian B. Pour-El and J. Ian Richards, *Computability in analysis and physics*, Perspectives in Mathematical Logic, Springer-Verlag, Berlin, 1989. MR 1005942 (90k:03062)

[sta] *Stack overflow*, http://stackoverflow.com/.

[Wei00] Klaus Weihrauch, *Computable analysis*, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag, Berlin, 2000, An introduction. MR 1795407 (2002b:03129)