Inhaltsverzeichnis

0 Introduction

We aim to implement a type of analytic functions in iRRAM. The iRRAM is a C++ library for arbitrary precision real analysis. We will begin by giving a short overview over iRRAM.

1 Computable analytic functions

2 iRRAM

At first glance the most reasonable approach to computable analysis would be to represent a real number x by an algorithm P taking a natural number n and returning a approximation, i.e. a appropriately encoded rational number x_n such that $|x - x_n| < 2^{-n}$. This kind of proceeding bears the following problems:

- 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 of any Program and often to uncontrollable growth of memory consumption.
- A 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, since it can lead to pathological behavior.

Thus the iRRAM chooses a different approach. Namely the real numbers are represented by finite Intervals containing said real number. All manipulations are carried out with these intervals. If a situation occurs, where the precision is simply not sufficient anymore, the whole calculation is restarted with higher precision. We call this procedure a *reiteration*.

iRRAMs approach does not suffer the problems listed above, but brings its own (which are hopefully more manageable):

- 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 reevaluated. 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.

Another point is, that programs for iRRAM are written in C++. Since C++ is a very powerful programing language, it is often possible for the user to do things he is not really supposed to do. We will encounter this problem ourselves in **??**.

We will now consider some parts of iRRAM more closely.

2.1 iRRAM functions

We did already mention, that output in iRRAM can be a problem and that one should use the output methods provided by iRRAM. To cope with the difficulties of duplicated output, iRRAM has its own class of streams iRRAM::orstream, which replaces the standard output streams. The following can be used for output:

- The iRRAM::orstream iRRAM::cout.
- The functions iRRAM::rwrite, iRRAM::rwritee and iRRAM::rshow.
- The function iRRAM::rfprint.

In the following we will leave the preceding iRRAM:: away. This coincides with the syntax used inside of iRRAM programs. One should note, that cout differs from std::cout: Pointers will be output as 1 if they point somewhere and as 0 if they do not. This is important, since pointer addresses can change in reiterations, which could lead to inconsistent output.

2.2 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.

2.3 Limits

3 Tools

In C++, there are four kinds of functional objects: functions, member functions, function pointers and member function pointers. Since the first two are very common, we will start at function pointers. It is important to note, that function pointers, in contrast to regular pointers, do not point to a chunk of memory where a function is located, but to a the piece of code where the function is defined. 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 actually solve this problem, since classes, and with them their member functions, can be dynamically created and destroyed. Really using member functions would be very involved, luckily for us C++11 added improved syntax for exactly this.

We will in the following sections review the tools we need to handle functions and their dependencies.

3.1 std::functions

The std::function template is a general-purpose polymorphic function wrapper (according to cppreference.com). A `std::function` can be defined by

```
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:

```
#include<functional>

using std::function;

double f(int i) {
  return double(i);
}

int main()
{
  function<double(int)> g = f;
  cout << g(4) << endl;
  return 0;
}
```

The std::function can do a lot more than regular functions though, as we will see in the coming chapters.

The syntax `function<RESULT(PARAM)>` of `std::functions` 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:

```
1 template<class T>
2 class function {};
3
4 template<class T, class S>
5 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.

3.2 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

```
1 [/*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 and can be accessed from the algorithm anyway. Variables may be captured by copy or, if they are preceded by an '`&`', by reference.

`/*parameters*/` is 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:

```
1 int i = 5;
2 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 reason is the following: 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.

3.3 `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 is already destroyed. This will lead to an error.

4 putting stuff together

4.1 the class FUNC

4.2 the class POWERSERIES

4.3 the class BASE_ANALYTIC